Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DEL SOFTWARE

SparkEC: Reingeniería y optimización de una herramienta Big Data para la corrección de errores en conjuntos de datos genéticos

Estudiante: Marco Martínez Sánchez
Directores: Roberto Rey Expósito
Juan Touriño Domínguez

A Coruña, junio de 2020.

A mis padres, a mi hermano, a mis amigos, y a Paula; por haber estado a mi lado en todo el viaje que ha sido llegar hasta aquí.

Agradecimientos

En primer lugar me gustaría agradecer a toda mi familia, por haberme apoyado y acompañado a lo largo de todos estos años de estudio, y de este proyecto. También, me gustaría agradecer especialmente a mis directores, ya que fue gracias a su orientación, consejo y dirección que ha sido posible llevar a cabo este trabajo. Muchas gracias a todos los que habéis contribuido a que este proyecto fuera posible.

Resumen

Con el presente Trabajo de Fin de Grado (TFG) se plantea el rediseño y reimplementación de la herramienta paralela CloudEC con el objetivo último de obtener una mejora de su rendimiento en entornos clúster. CloudEC permite realizar la corrección de errores en secuencias de ADN para así incrementar la calidad de las bases leídas en el proceso de secuenciación. Tanto la herramienta original (CloudEC) como la nueva desarrollada en este TFG (SparkEC) están enfocadas al manejo de grandes volúmenes de datos haciendo uso de frameworks de procesamiento Big Data. En el caso de CloudEC está implementada con Apache Hadoop y, en el caso de la nueva herramienta fruto del resultado de este proceso de reingeniería, se ha seleccionado Apache Spark. Ambos frameworks son gratuitos, de código abierto y ampliamente utilizados tanto en investigación como en la industria.

La herramienta ha sido desarrollada utilizando prácticas bien asentadas en el ecosistema de la Ingeniería del Software. Para realizar el diseño, si bien se ha conservado la arquitectura subyacente del sistema original, esta ha sido refinada utilizando patrones de diseño y arquitecturales, buscando con ello siempre una mejora en la mantenibilidad y extensibilidad del software. Adicionalmente, SparkEC ofrece funcionalidades y configuraciones adicionales. Con ellas, se espera que los usuarios puedan adecuar mejor cada ejecución a sus datos de entrada y al hardware disponible para realizar la computación.

Respecto al rendimiento, se ha realizado una extensa evaluación experimental con distintos conjuntos de datos, configuraciones y versiones del software, comparándolas entre sí para analizar el impacto de las optimizaciones propuestas, y con respecto a la herramienta original para conocer la aceleración obtenida en cada escenario.

La herramienta desarrollada en este TFG se encuentra disponible para su descarga en el siguiente repositorio Git: `https://github.com/mscrocker/SparkEC`.

Abstract

This BSc Thesis proposes the redesign and reimplementation of the parallel tool named CloudEC in order to improve its performance in cluster environments. CloudEC allows performing error correction over DNA reads to increase the quality of the base pairs obtained during the sequencing process. Both the original tool (CloudEC) and the one developed in this project (SparkEC) are focused on handling large datasets by relying on Big Data processing frameworks. In the case of CloudEC it is implemented upon Apache Hadoop, whereas Apache Spark has been selected for the new tool developed as the result of the reengineering

process. Both Hadoop and Spark are free, open-source frameworks widely used in research and industry.

The tool has been developed following practices well settled in the Software Engineering ecosystem. In order to perform the design, the underlying architecture of the original system has been maintained but refined using design and architectural patterns, always trying to improve the maintainability and extensibility of the software. Furthermore, SparkEC provides additional features and settings. With them, it is expected that users are able to better fit each execution to their input datasets and also to the available hardware for performing the computations.

Regarding performance, an extensive experimental evaluation using different datasets, settings and versions of the software has been carried out, making comparisons among them to analyze the impact of the optimizations proposed, and comparing them to the original tool in order to measure the acceleration obtained in each scenario.

The tool developed in this project is publicly available to download at the following Git repository: https://github.com/mscrocker/SparkEC.

Palabras clave:

- Big Data
- Genómica
- NGS
- k-mer
- Apache Hadoop
- Apache Spark
- HDFS

Keywords:

- Big Data
- Genomics
- NGS
- k-mer
- Apache Hadoop
- Apache Spark
- HDFS

Índice general

Índice de figuras

Índice de tablas

Listados

Capítulo 1

Introducción

En este primer capítulo introductorio se muestra la principal motivación que aporta interés al desarrollo de este Trabajo de Fin de Grado (TFG), los objetivos que se proponen cumplir con él y se presentan brevemente algunos trabajos relacionados con este proyecto. Por último, se detalla la estructura del documento.

1.1 Motivación

En la actualidad resulta necesario llevar a cabo, principalmente con fines médicos y de investigación, la secuenciación de un volumen creciente de secuencias genéticas. Este proceso típicamente implica llevar a cabo la extracción de una muestra de ADN, la aplicación de una serie de transformaciones sobre estas muestras, y finalmente la obtención de las secuencias mediante un secuenciador. Sin embargo, estos secuenciadores no tienen una fiabilidad absoluta, introduciendo errores en las bases leídas que a menudo son anotadas con un valor para representar su fiabilidad.

Un preprocesamiento habitual en muchos pipelines bioinformáticos es la corrección de errores en conjuntos de datos genómicos. Existen múltiples herramientas y algoritmos que son capaces de hacer un análisis de las secuencias genéticas, guiándose por los valores de las calidades de las bases, y aplicar correcciones sobre ellas en base a esta información. Entre las herramientas paralelas existentes se encuentra CloudEC [1], la cual hace uso del framework Big Data de código abierto Apache Hadoop [2] para realizar estas correcciones en entornos distribuidos como clústeres y plataformas Cloud.

Por otro lado, debido a la continua evolución de las tecnologías de secuenciación en la llamada Next-Generation Sequencing (NGS) [3], se ha producido un aumento considerable del volumen de datos genéticos disponibles junto con una reducción drástica de su coste. Esto hace manifiesta la necesidad de aplicar estrategias y tecnologías propias del ámbito Big Data para procesar y analizar esta gran cantidad de información en tiempos razonables.

1.2 Objetivos

El objetivo principal de este TFG es el desarrollo de SparkEC, una herramienta paralela para la corrección de errores en conjuntos de datos genómicos que sea distribuida, escalable y eficiente. Con este cometido, se tomará como base el algoritmo de corrección propuesto por CloudEC, reimplementándolo con una nueva arquitectura sobre el framework Apache Spark [4]. Mediante estos cambios, se pretende reducir el impacto en tiempo que esta fase de corrección pueda tener sobre el preprocesado de los datos genéticos para su posterior análisis.

Además, se buscará proporcionar un software más genérico, configurable y mantenible que el original, aplicando para ello metodologías propias de la Ingeniería del Software a su diseño e implementación.

1.3 Trabajo relacionado

Existen múltiples herramientas que son capaces de llevar a cabo la corrección de errores en conjuntos de secuencias genéticas. Este proyecto se enfoca en aquellas que pertenecen a la familia de algoritmos basados en alineamientos múltiples de secuencias o MSA (Multiple Sequence Alignment) [5]. Entre ellas podemos destacar la herramienta ALLPATHS-LG [6], la cual proporciona múltiples utilidades para la manipulación de secuencias entre las que se encuentra una enfocada a la corrección de errores. Se trata de una solución multithread implementada en C++ que es capaz de proporcionar buen rendimiento en entornos de memoria compartida. Sin embargo, no soporta entornos distribuidos, lo cual limita su escalabilidad para manejar grandes volúmenes de datos.

Por otro lado, CloudRS [7] toma como base la implementación del método de corrección de ALLPATHS-LG, tratando de aplicar optimizaciones sobre él e implementándolo en Java haciendo uso del framework Apache Hadoop. Así, CloudRS es capaz de funcionar de forma distribuida sobre un clúster o una plataforma Cloud, consiguiendo una aceleración sobre el sistema original al ser capaz de aprovechar más de un nodo.

Por último, CloudEC [1] se considera una evolución de CloudRS introduciendo un nuevo algoritmo de corrección (SpreadCorrect) que permite incrementar la calidad de las correcciones así como obtener un rendimiento superior. Sin embargo, a pesar de que consigue superar a su predecesor, CloudEC sigue presentando un desempeño excesivamente bajo en algunos escenarios, derivado de su implementación con Hadoop y su modelo de procesamiento basado en disco, el cual es altamente dependiente del uso de memoria secundaria.

1.4 Estructura de la memoria

A continuación, se muestran los capítulos que componen esta memoria, explicando brevemente su contenido.

- ***Introducción:*** con este capítulo se brinda al lector una visión general del contexto en el que se enmarca este TFG, así como de los aspectos que serán tratados a lo largo del documento.
- ***Conceptos previos:*** este capítulo introduce los conceptos teóricos previos necesarios sobre el campo de la Bioinformática y el ámbito Big Data, pilares sobre los que se asienta el proyecto.
- ***Tecnologías y herramientas:*** este capítulo ofrece información acerca de las diferentes tecnologías y herramientas utilizadas para acometer el desarrollo del TFG.
- ***Metodologías de desarrollo:*** en este capítulo se encuentra información acerca de las distintas metodologías de desarrollo que han sido utilizadas para el desarrollo de SparkEC.
- ***Diseño y desarrollo:*** este capítulo describe los principales aspectos del diseño e implementación de la herramienta, incluyendo información sobre las tareas y la planificación de los Sprints que fueron necesarios para completar el desarrollo.
- ***Evaluación experimental:*** en este capítulo se muestran y analizan los principales resultados de la evaluación de rendimiento de SparkEC en un entorno clúster, comparándola con la herramienta CloudEC.
- ***Conclusiones y trabajo futuro:*** el último capítulo presenta las conclusiones del trabajo tras finalizar el desarrollo y evaluación de la herramienta. También se proponen distintas líneas de mejora como posible trabajo futuro.

Capítulo 2

Conceptos previos

En este capítulo se pretenden cubrir aquellos conceptos teóricos sobre los que fue necesario documentarse previamente para llevar a cabo la realización del proyecto, y que resultan de interés destacar para contextualizar y comprender el desarrollo posterior de la herramienta.

2.1 Bioinformática

La Bioinformática se puede definir como la disciplina consistente en la aplicación y puesta a disposición de la Informática a la Biología. Esta ciencia se puede aplicar para resolver multitud de problemas tanto a nivel teórico, dando soporte a tareas de investigación biológicas, como a un nivel práctico, asistiendo procesos de la Medicina.

2.1.1 Next-Generation Sequencing

Las técnicas de secuenciación de segunda generación o NGS [3] se definen como el conjunto de tecnologías destinadas a llevar a cabo la secuenciación masiva a gran escala de cualquier ácido nucleico con el objetivo de obtener y analizar grandes volúmenes de datos genómicos. Esta segunda generación permitió, gracias a las mejoras tecnológicas introducidas, acelerar enormemente el proceso de secuenciación y además abaratar significativamente los costes de todo el proceso. Así, mediante el uso de tecnologías NGS es posible en la actualidad secuenciar el genoma humano completo en tan solo un día o incluso menos, mientras que con las tecnologías anteriores fue necesaria más de una década para lograrlo.

Dentro de este marco, la Bioinformática ha hecho importantes aportaciones a todo el proceso que se realiza posteriormente a la secuenciación, ofreciendo diferentes herramientas para el manejo, edición, análisis o almacenamiento de las secuencias. Entre estas herramientas se encuentran conversores de ADN a ARN y viceversa, operaciones de filtrado y recortado, o incluso algoritmos capaces de corregir errores en las bases que se pudieran haber introducido durante el proceso de secuenciación.

```
>SEQUENCE_1
MTEITAAMVKELRESTGAGMMDCKNALSETNGDFDKAVQLLREKGLGKAAKKADRLAAEG
LVSVKVSDDFTIAAMRPSYLSYEDLDMTFVENEYKALVAELEKENEERRRLKDPNKPEHK
IPQFASRKQLSDAILKEAEEKIKEELKAQGKPEKIWDNIIPGKMNSFIADNSQLDSKLTL
MGQFYVMDDKKTVEQVIAEKEKEFGGKIKIVEFICFEVGEGLEKKTEDFAAEVAAQL
>SEQUENCE_2
SATVSEINSETDFVAKNDQFIALTKDTTAHIQSNSLQSVEELHSSTINGVKFEEYLKSQI
ATIGENLVVRRFATLKAGANGVVNGYIHTNGRVGVVIAAACDSAEVASKSRDLLRQICMH
```

Listado 2.1: Ejemplo de secuencias en formato FASTA

2.1.2 Formatos de secuencias

Para representar y almacenar las secuencias genéticas y poder trabajar con ellas desde el campo de la Bioinformática se han definido una serie de formatos, que permiten representar tanto a las propias secuencias, como a algunos metadatos obtenidos durante el proceso de secuenciación. Estos formatos generalmente suelen estar basados en texto plano, pero se suelen evitar el uso de caracteres especiales. Así, generalmente basta con utilizar caracteres ASCII para representar tanto las secuencias como los metadatos. A continuación, se explican dos formatos de representación posibles de estas secuencias genéticas que son ampliamente utilizados en Bioinformática.

- ***FASTA:*** este formato permite representar secuencias de ácidos nucleicos (i.e. ADN/ARN) y péptidos en el que los pares de bases o los aminoácidos se representan usando códigos de una única letra. Así, este formato representa cada lectura utilizando varias líneas de texto. La primera línea, que va precedida por el carácter '>', indica el identificador único de la secuencia. A partir de ahí, se introducen una o varias líneas que representan en codificación ASCII los nucleótidos o aminoácidos (ver Listado 2.1).

- ***FASTQ:*** de forma similar al anterior, este formato permite representar secuencias de nucleótidos que se asocian a un identificador único. Sin embargo, FASTQ incluye información adicional a las bases. Entre estos campos adicionales se encuentran: la fiabilidad o calidad de cada base proporcionada por el secuenciador, y opcionalmente un comentario para la lectura. Para ello hace uso de cuatro líneas de texto para representar cada secuencia (ver Listado 2.2). En la primera línea se representa el identificador de la secuencia precedido por el prefijo '@'. En la segunda línea se representan las bases mientras que la tercera incluye el comentario precedido por el carácter '+' (en muchas ocasiones, se duplica el identificador en este comentario). Finalmente, la última línea proporciona las calidades de las bases, en la que el carácter n-ésimo de calidad se corresponde con la fiabilidad (o calidad) de la base n-ésima.

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Listado 2.2: Ejemplo de secuencia en formato FASTQ

2.1.3 Algoritmos de corrección

Entre las diferentes aplicaciones de la Bioinformática se mencionó previamente la existencia de algoritmos de corrección. La herramienta que se pretende desarrollar como producto de este TFG se enmarca en este ámbito de sistemas de corrección que permiten detectar y proponer cambios para corregir errores introducidos durante la secuenciación.

Como se mencionó en la Sección 1.1, la existencia de estos algoritmos se debe a que el proceso de secuenciación no es perfecto, y en ocasiones introduce errores. Los errores que se suelen producir pueden ser de tres tipos diferentes:

- ***Errores de eliminación:*** en los que una base que se encontraba en la secuencia original es eliminada de la lectura.
- ***Errores de adición:*** en los que se añade una base adicional a la lectura, que no existía en la secuencia original.
- ***Errores de sustitución:*** en los que, si bien se detecta la existencia de una base correctamente, la base detectada es diferente a la base original. En algunos casos se detecta la existencia de una base, pero no es posible determinar qué base es, siendo representada con el carácter 'N' en los formatos FASTA/FASTQ.

Los algoritmos de corrección analizados en la Sección 1.3 están especializados en llevar a cabo correcciones de errores de sustitución, ya que suelen ser el tipo de error más frecuente en las nuevas tecnologías de secuenciación. De acuerdo a la literatura [5], los algoritmos se suelen clasificar fundamentalmente en tres grupos diferenciados:

- ***K-Spectrum:*** estos algoritmos se basan en que subsecuencias similares encontradas repetidamente en múltiples lecturas pueden pertenecer a la misma posición en el genoma. Basándose en esta idea general, los algoritmos descomponen cada lectura en un conjunto de subsecuencias de longitud K, llamadas k-mers, y suelen contar el número de apariciones de cada k-mer a lo largo de todo el conjunto de datos, construyendo así el denominado espectro de k-mers. De esta forma, pueden clasificar los k-mers en sólidos (si tienen un número de apariciones superior a un determinado umbral) y débiles (si no alcanzan el umbral). Tras esto, se trata de aplicar un reducido número de correcciones

sobre los k-mers débiles para convertirlos en k-mers sólidos. Estas modificaciones sobre los k-mers se traducen finalmente en correcciones sobre las secuencias.

- ***Suffix Tree / Array:*** estos algoritmos se consideran una generalización del caso anterior. En este caso se generan k-mers para distintas distancias de Hamming, y se aplican correcciones para ellas.

- ***MSA:*** en este caso, los algoritmos buscan k-mers iguales en diferentes lecturas, y se usan estos k-mers para realizar alineamientos. Estos sistemas permiten alinear así, utilizando los k-mers de forma auxiliar, las diferentes lecturas para después obtener una lectura de referencia y aplicar correcciones para aquellas lecturas que no sigan la lectura base.

Cabe resaltar que, tanto la herramienta de partida (CloudEC) como la desarrollada en este TFG (SparkEC), se enmarcan dentro de la categoría MSA. Debido a que estos algoritmos realizan múltiples alineamientos durante el proceso de corrección, suelen ser los más costosos en términos computacionales.

2.2 Big Data

El término Big Data hace referencia a un conjunto de tecnologías, técnicas y herramientas diseñadas específicamente para almacenar, procesar y analizar volúmenes de datos lo suficientemente grandes como para que no puedan ser tratados mediante métodos convencionales. El Big Data se encuentra actualmente en auge, debido al aumento de información que se genera en Internet, las redes sociales o dispositivos IoT, entre otros, y a la necesidad creciente de analizar ese gran volumen de datos para extraer información útil.

En la actualidad existen multitud de soluciones diseñadas específicamente para trabajar con estos grandes volúmenes de datos, entre las que se encuentran tecnologías para almacenamiento de datos, o para su procesamiento tanto en tiempo real como en lotes o *batches*.

2.2.1 Paradigma MapReduce

El paradigma MapReduce fue una de las primeras soluciones que se propusieron para el tratamiento de grandes volúmenes de datos. Este paradigma, originalmente propuesto por Google [8], cuenta con una implementación en C++ propietaria que inspiró la aparición de sistemas posteriores, como por ejemplo el caso de Hadoop de la fundación Apache. Mediante este mecanismo se facilita la implementación de soluciones tecnológicas que requieran realizar cómputo de forma distribuida en entornos clúster/Cloud.

Una piedra angular de este modelo de procesamiento es contar con un sistema de ficheros distribuido. De este modo, es posible llevar a cabo la computación en los mismos nodos que

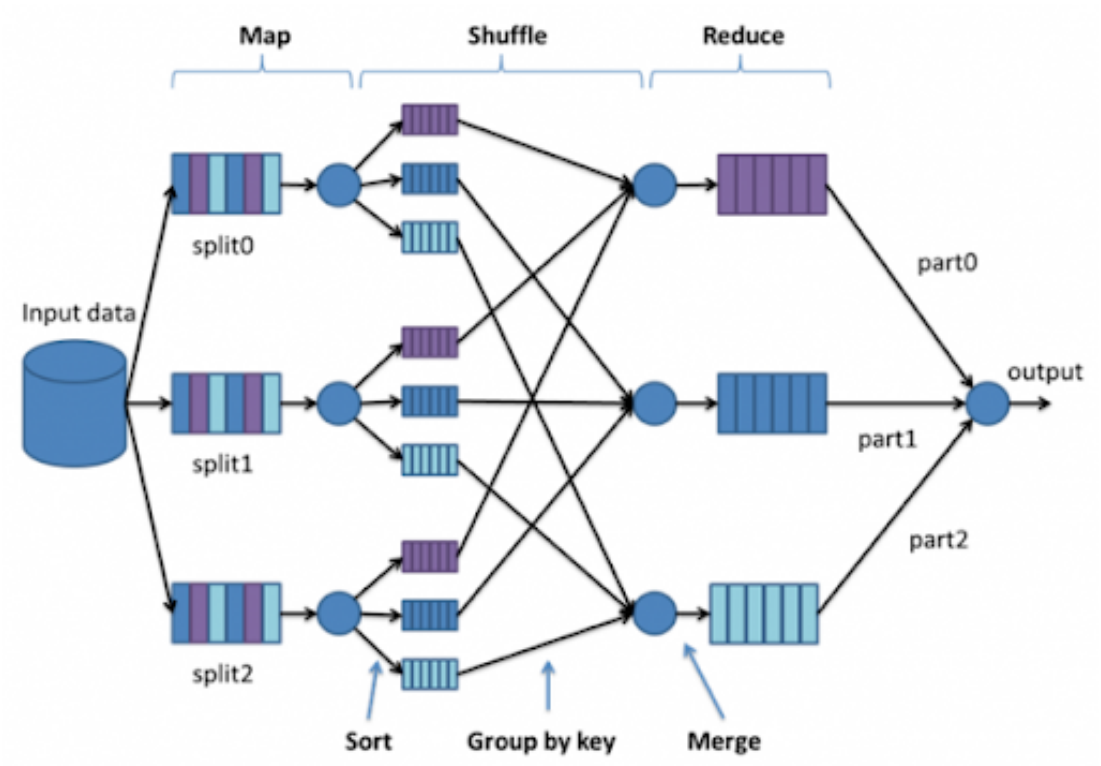

Figura 2.1: Paradigma MapReduce

contienen los datos, consiguiendo una aceleración en el procesamiento. Es por ello que de la mano de la implementación MapReduce de Google surgió también el sistema de ficheros GFS (Google File System) [9]. Este sistema de ficheros distribuido funciona mediante una arquitectura líder-trabajador en la que los datos se encuentran repartidos entre los trabajadores (i.e. nodos de un clúster). Igual que MapReduce inspiró el desarrollo de nuevos sistemas como Hadoop, tomando GFS como base surgieron otros sistemas de ficheros como HDFS (Hadoop Distributed File System) [10].

El modelo MapReduce permite procesar y analizar conjuntos de datos en los que existen gran cantidad de tuplas individuales, generando otros nuevos conjuntos de datos tras llevar a cabo las operaciones necesarias. Para ello, este paradigma se basa en la existencia de tres fases muy diferenciadas, de las cuales dos son programables, y que se pueden ejecutar tantas veces como sea necesario. Estas fases, mostradas en la Figura 2.1, son:

- ***Map:*** es una de las dos fases programables del modelo. En ella, para cada tupla o par clave-valor de entrada se ejecuta una función especificada por el programador. Esta función puede devolver uno o varios pares clave-valor de salida (se permite también que no devuelva ninguno).

- ***Reduce:*** esta es la segunda fase programable en la que para cada clave distinta emitida en la fase *Map* (y el conjunto de valores que compartían esta clave), se ejecuta la función especificada por el programador. En este caso, la función debe reducir todos estos valores a un único par clave-valor de salida.
- ***Shuffle and Sort:*** esta fase no programable e intermedia se sitúa entre las fases *Map* y *Reduce*. En ella se produce una redistribución de los datos entre los nodos del clúster disponibles, consiguiendo de este modo agrupar aquellos valores que contengan una misma clave, para poder posteriormente ejecutar la fase *Reduce*.

Capítulo 3

Tecnologías y herramientas

En este capítulo se describen brevemente las tecnologías concretas que se utilizaron para llevar a cabo el desarrollo de la herramienta SparkEC. Entre ellas se encuentran implementaciones específicas de los paradigmas teóricos mencionados en el capítulo anterior, así como utilidades de apoyo al desarrollo de aplicaciones Java.

3.1 Tecnologías Big Data

Para comenzar, en esta sección se mencionan las tecnologías enfocadas al Big Data que hacen posible el funcionamiento de las herramientas SparkEC y CloudEC.

3.1.1 Apache Hadoop

Apache Hadoop [2] es un framework de procesamiento distribuido escrito completamente en Java y orientado al almacenamiento y análisis de grandes volúmenes de datos. Hadoop implementa en Java el paradigma MapReduce de Google, que se basa en la existencia de tres fases diferenciadas: la fase *Map*, la fase *Reduce* y la fase *Shuffle and Sort*, explicadas previamente en la Sección 2.2.1.

Además, Hadoop hace uso del sistema de ficheros distribuido HDFS y, en sus versiones más recientes, del gestor de recursos YARN (Yet Another Resource Negotiator) [11].

3.1.2 Hadoop Distributed File System

HDFS [10] es un sistema de ficheros distribuido implementado en Java, inspirado en el sistema de ficheros GFS de Google, tolerante a fallos, escalable y eficiente.

Los ficheros almacenados en HDFS se dividen en diferentes bloques de tamaño fijo y configurable por el usuario, que se distribuyen por los nodos que forman el clúster Hadoop. De este modo, cuando se ejecuten las operaciones de cómputo en los nodos, los datos ya se encontrarán repartidos por el clúster, permitiendo así su procesamiento distribuido y obteniendo

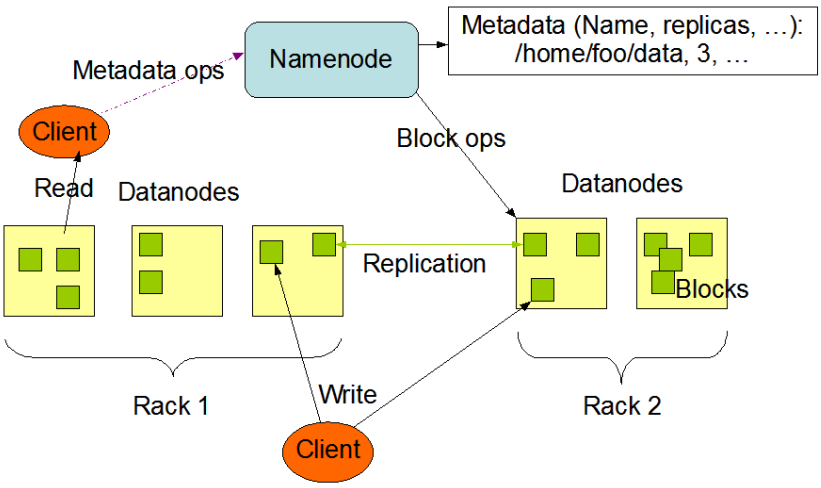

Figura 3.1: Arquitectura de HDFS

una gran mejora en el rendimiento. Además, los bloques pueden replicarse en múltiples nodos del clúster con el objetivo de proporcionar tolerancia a fallos a nivel del sistema de ficheros.

HDFS basa su diseño en dos componentes principales que se muestran en la Figura 3.1:

- ***NameNode:*** este componente se encarga de gestionar los metadatos y conocer en qué nodos se encuentran cada uno de los bloques que conforman los ficheros almacenados en HDFS.
- ***DataNode:*** este componente permite almacenar y proveer los bloques de los ficheros que se almacenan en un determinado nodo del clúster.

3.1.3 Apache YARN

YARN [11] es un gestor de recursos distribuido desarrollado por la fundación Apache dentro de la familia de tecnologías Hadoop. Surgió como una respuesta a la necesidad de contar con un sistema que permitiese gestionar los recursos hardware disponibles para diferentes tareas ejecutadas de forma distribuida.

YARN tiene una arquitectura que permite aislar la gestión y monitorización de recursos de la gestión de las aplicaciones. Así, se define el componente *ResourceManager*, que se encarga de gestionar todos los recursos del clúster, y el componente *NodeManager*, que se ejecuta en cada uno de los nodos del clúster, para recibir comandos y reportar su estado al *ResourceManager* (ver Figura 3.2). Por otro lado, YARN define los *ApplicationMaster*, que son los componentes encargados de coordinar la ejecución de una aplicación sobre el clúster. Son, por tanto, dependientes del framework que se utilice, y no se basan necesariamente en el modelo MapReduce.

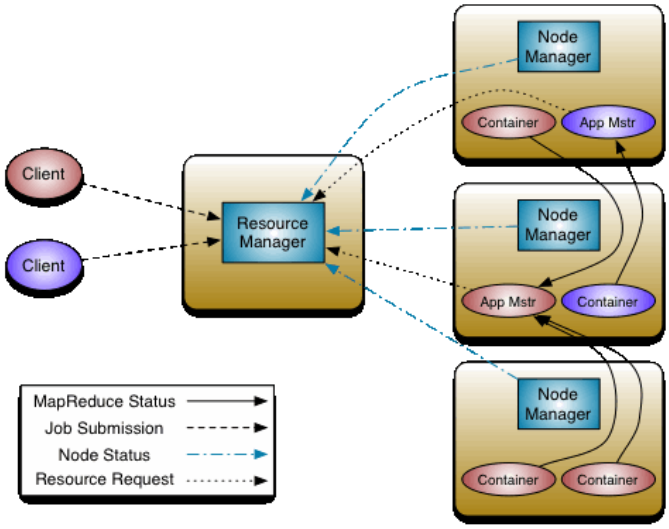

Figura 3.2: Arquitectura de YARN

Así, mediante estos componentes es posible la interacción entre las aplicaciones y los otros servicios de YARN.

3.1.4 Apache Spark

Apache Spark [4] es, de forma similar a Hadoop, un framework de procesamiento distribuido desarrollado por la fundación Apache que permite procesar grandes volúmenes de datos. Spark está escrito en el lenguaje de programación Scala, aunque dispone de APIs que permiten utilizarlo con otros lenguajes como Java, Python o R.

Por otro lado, Spark soporta muchas de las tecnologías preexistentes proporcionadas por el ecosistema Hadoop. Por tanto, como sistema de ficheros puede utilizar HDFS y también YARN como gestor de recursos (aunque también soporta otros sistemas de ficheros y gestores).

Este framework provee además diferentes componentes que permiten extender las capacidades y el rendimiento de Hadoop. Entre estos componentes adicionales (ver Figura 3.3), podemos destacar los siguientes:

- ***Spark SQL:*** permite realizar consultas de tipo SQL sobre una base de datos relacional, ya sea mediante Apache Hive, ODBC, JDBC u otras.

- ***Spark Streaming:*** este componente se enfoca en el procesamiento de flujos de datos o streams. En este caso se pueden definir flujos de cálculo que procesen un volumen de datos significativamente menor, obteniendo resultados también en un menor tiempo, incluso en tiempo real.

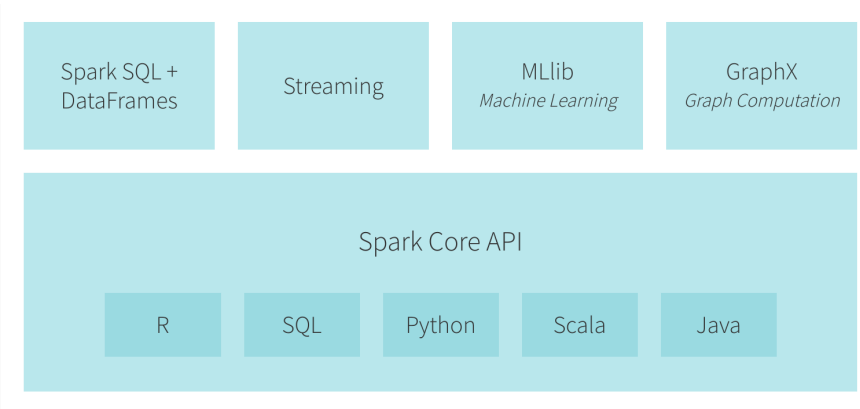

Figura 3.3: Componentes de Spark

- ***MLib:*** en este caso, este módulo proporciona diferentes algoritmos de aprendizaje automático o *machine learning*, facilitando así la implementación distribuida de soluciones basadas en inteligencia artificial.
- ***GraphX:*** este módulo permite realizar cálculos paralelos sobre grafos.

Respecto al funcionamiento de *Spark Core*, el módulo base que hace funcionar todo el resto de componentes, se basa en una estructura de datos conocida como RDD (Resilient Distributed Dataset) [12, 13]. Esta estructura consiste en una colección de datos que se encuentra distribuida por los diferentes nodos del clúster Spark, y que cuenta con la propiedad de ser inmutable (las operaciones que se ejecuten sobre él devuelven un nuevo RDD). Así, un RDD consta de numerosas particiones de datos, almacenándose cada una de ellas en, al menos, un nodo (aunque Spark puede almacenar datos redundantes para tener la capacidad de recuperarse ante errores). Sobre estos RDDs es posible llevar a cabo operaciones, que pueden provocar o no operaciones de shuffle. Fundamentalmente, las operaciones sobre un RDD se pueden clasificar en dos tipos:

- ***Transformaciones:*** este tipo de operaciones permite transformar los datos. Se ejecutan de modo *lazy*, por lo que son llamadas no bloqueantes para una aplicación Spark. No modifican los datos en el RDD original, sino que devuelven un nuevo RDD con el resultado de la operación. Cabe destacar que Spark almacena el orden en el que fueron ejecutadas las transformaciones, de modo que en caso de error es capaz de recalcular particiones o RDDs concretos, simplemente volviendo a recalcular las transformaciones aplicadas sobre ellos.
- ***Acciones:*** estas operaciones se diferencian de las anteriores en que se ejecutan en modo *eager*, por lo que sí son llamadas bloqueantes en este caso. Generalmente permiten obtener información del RDD sobre el que se ejecutan, como puede ser solicitar su número

de elementos o simplemente provocar que los datos sean persistidos a disco. También es importante tener en cuenta que, ya que estas son llamadas bloqueantes a diferencia de las transformaciones, una llamada a una acción es la que desencadena la ejecución de las transformaciones pendientes hasta ese momento.

También es importante destacar que en Spark los RDDs son eliminados, total o parcialmente, en base a un algoritmo LRU (Least Recently Used). En caso de que un RDD volviera a utilizarse puede recomputarse ya que Spark conoce el orden de ejecución de las transformaciones, como ya se mencionó. Sin embargo, para mejorar el rendimiento de las aplicaciones, Spark permite que los RDDs puedan ser persistidos con diferentes niveles. Persistir (o cachear) un RDD permite almacenarlo para utilizarlo posteriormente en otras operaciones. Spark nos ofrece los siguientes niveles de persistencia para un RDD:

- ***MEMORY_ONLY:*** el RDD se almacena únicamente utilizando la memoria principal. Si no cabe, se eliminarían particiones, las cuales necesitarían recalcularse en caso de volver a ser necesarias.
- ***MEMORY_AND_DISK:*** al igual que en *MEMORY_ONLY*, el RDD se almacena en memoria principal. Sin embargo, en este caso se volcarían las particiones a almacenamiento persistente en caso de no disponer de memoria suficiente.
- ***MEMORY_ONLY_SER:*** este nivel funciona igual que *MEMORY_ONLY*, pero almacenando los elementos del RDD de forma serializada, consiguiendo reducir el uso de memoria pero aumentando el de CPU.
- ***MEMORY_AND_DISK_SER:*** el funcionamiento sería el mismo que el caso del nivel *MEMORY_AND_DISK*, pero almacenando los elementos de forma serializada en disco.
- ***DISK_ONLY:*** en este nivel las particiones existen únicamente en almacenamiento persistente.
- ***OFF_HEAP:*** este nivel se comportaría como *MEMORY_ONLY*, salvo que se utilizaría memoria fuera del heap que gestiona la máquina virtual de Java (*off-heap memory*).

Además, para todos los niveles de persistencia existen variantes que permiten forzar que cada partición se almacene por duplicado, incrementando así la tolerancia a fallos, pero consumiendo un mayor espacio.

Respecto al modelo de ejecución de Spark, sigue estando basado en el modelo MapReduce, por lo que sigue contando con las fases *Map*, *Reduce* y *Shuffle and Sort*. Lo que sucede en Spark es que estas fases no son controladas directamente por la aplicación, sino por el framework. Así, Spark va construyendo un grafo acíclico dirigido o DAG (*Directed Acyclic Graph*) con

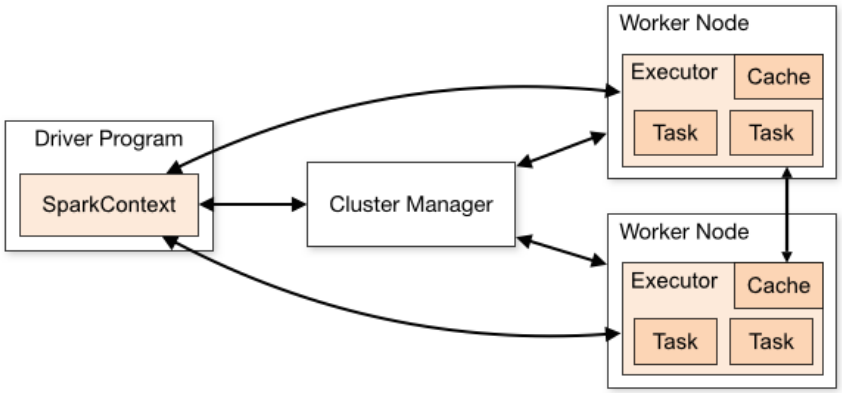

Figura 3.4: Arquitectura de Spark en modo distribuido

todas las transformaciones que se realizan sobre un RDD, hasta que se realiza una acción, provocando que el grafo se ejecute.

De este modo, Spark representa las operaciones *Shuffle and Sort* como enlaces en el grafo, y trata de agrupar todo lo que sea posible las transformaciones en los nodos (de modo que un nodo del grafo puede contener varias operaciones). Así, cada nodo del grafo representaría un trabajo MapReduce, y para cada nodo del grafo y cada partición del RDD, se define una tarea que debe ejecutarse por un nodo trabajador del clúster.

Para que Spark sea capaz de ejecutar las operaciones definidas en los grafos, nos ofrece dos modos de ejecución diferentes:

- ***Modo local:*** en este modo de ejecución Spark hace uso de los recursos computacionales de únicamente un nodo, en el que se ejecuta tanto el código de la aplicación que define el grafo de tareas (denominado *driver*) , como las propias tareas de procesamiento en sí.
- ***Modo distribuido:*** en este modo Spark se ejecuta en un clúster o Cloud que gestiona mediante la arquitectura líder-trabajador. Existe un nodo que ejecuta el código del *driver*, mientras que el resto de nodos trabajadores realizan el procesamiento de las tareas (ver Figura 3.4).

3.1.5 Hadoop Sequence Parser

HSP (Hadoop Sequence Parser) [14] es una biblioteca Java desarrollada en el GAC (Grupo de Arquitectura de Computadores de la Universidade da Coruña) que permite la lectura de secuencias genéticas almacenadas en HDFS. Entre los diferentes formatos que soporta se encuentran FASTQ y FASTA.

HSP ha sido integrada en la herramienta desarrollada en este TFG para permitir realizar la lectura de los datos en el formato FASTQ.

3.1.6 BDEv

BDEv (Big Data Evaluator) [15, 16] es una herramienta desarrollada en el GAC que asiste en la evaluación de rendimiento de diferentes frameworks Big Data.

Entre otras funcionalidades, BDEv permite agilizar la configuración de diferentes entornos de prueba, definiendo una configuración por defecto y configuraciones específicas para cada experimento a realizar. Gracias a que esta herramienta soporta tanto Hadoop como Spark, entre otros frameworks, fue posible llevar a cabo la evaluación de rendimiento de SparkEC y de CloudEC en un entorno clúster sin necesidad de desplegar manualmente los frameworks.

Mediante BDEv se puede configurar automáticamente el entorno para hacer uso de las configuraciones de HDFS, YARN, Hadoop y Spark de una forma óptima. También se hace cargo de la configuración hardware (v.gr. CPU, memoria), y configura el uso de las interfaces de red más adecuadas, además de integrarse de forma transparente con el gestor de recursos disponible en el clúster para conocer los nodos disponibles en cada experimento.

3.2 Herramientas de desarrollo

En esta sección se detallarán las diferentes herramientas que se han utilizado como soporte para poder llevar a cabo el desarrollo del proyecto con la mayor calidad posible.

3.2.1 Java

Java es un lenguaje de programación orientado a objetos, así como una API y una máquina virtual muy popular para el desarrollo de aplicaciones de diferentes propósitos. Este lenguaje de programación se compila a un lenguaje intermedio (bytecode) que es interpretado por una máquina virtual. De este modo, Java consigue independencia entre máquinas y sistemas operativos, así como aislamiento y facilidad de uso para los programadores, eliminando del desarrollo algunas fuentes de problemas frecuentes como la gestión explícita de la memoria.

Las APIs de Java han alcanzado a lo largo de los años una gran popularidad gracias a que cuentan con una gran abundancia de clases y métodos que agilizan la introducción de funcionalidades comunes. Así, para cada uno de los distintos objetivos que es necesario cumplir, se ofrece una API distinta. Además, este lenguaje de programación cuenta con un mecanismo de gestión de memoria automático e implícito. Esto es, el programador no debe preocuparse de realizar por él mismo y de forma explícita la gestión de la memoria, ya que la máquina virtual de Java cuenta con un recolector de basura que automáticamente buscará y eliminará todos aquellos objetos que ya no estén siendo referenciados.

Java fue seleccionado como el lenguaje de desarrollo para este proyecto de entre todas las opciones disponibles en Spark, teniendo en cuenta que este framework soporta otros lenguajes como Python, R o Scala como ya se mencionó en la Sección 3.1.4. Se tomó esta decisión ya

que la herramienta de base (CloudEC) estaba también escrita en Java, lo que facilitaba la reimplementación de la nueva herramienta. Por otro lado, había una mayor experiencia previa con este lenguaje que con los demás soportados en Spark, habiendo sido utilizado en varias asignaturas del Grado, por lo que escogerlo también agilizaba el desarrollo.

3.2.2 JUnit

JUnit es un sistema de pruebas automatizadas para software desarrollado en Java. Se enmarca dentro de la familia de tecnologías de prueba XUnit, y permite escribir y ejecutar pruebas Java conservando el estilo típico de las tecnologías de su familia.

JUnit estructura las pruebas en diferentes fases, permitiéndole al programador determinar lo que se hará en cada una de ellas. Así, es posible indicar código que sea ejecutado antes y después de las suites de pruebas, antes y después de cada prueba individual, y el código de cada una de las pruebas.

Esta herramienta fue seleccionada para realizar las pruebas de SparkEC, ya que se trata de una de las más utilizadas en el entorno Java, y también se tenía experiencia previa con ella.

3.2.3 Git

Git es un sistema de control de versiones descentralizado y distribuido que permite trabajar mediante el uso de etiquetas y ramas, llevando a cabo el desarrollo de forma paralela. Además, a diferencia de otros sistemas de versiones, se trata de una herramienta distribuida, lo que permite trabajar sin necesidad de disponer de un servidor central, haciendo uso de repositorios locales.

Git permite contar con ramas de desarrollo diferentes sin incrementar excesivamente el coste en almacenamiento del repositorio, ya que implementa las ramas como punteros. Gracias a esto, permite a los desarrolladores contar con un gran volumen de ramas en sus desarrollos, para controlar las versiones más fácilmente, y poder incluso paralelizar el desarrollo de las funcionalidades, asignándoles distintas tareas a cada uno de los miembros del equipo de desarrollo.

Fue publicado bajo la licencia GNU/GPL por Linus Torvalds en 2005 con el objetivo de mantener el kernel Linux. Actualmente, es uno de los sistemas de control de versiones más ampliamente utilizados en el mundo.

Por todas las facilidades que permite en la gestión de la configuración del proyecto, se utilizó Git como su sistema de control de versiones.

3.2.4 Eclipse

Eclipse es un IDE (*Integrated Development Environment*) multiplataforma desarrollado en Java. Se lanzó inicialmente en 2001, y disfruta de un gran número de usuarios, especialmente entre desarrolladores Java.

Eclipse está mantenido por la Eclipse Foundation sin ánimo de lucro, y se trata de un software libre y gratuito. Además, permite que su comunidad extienda las capacidades del IDE mediante el desarrollo de plugins, que permiten la integración con otras herramientas de desarrollo software, facilitándole así su trabajo a los desarrolladores.

Debido a la experiencia previa con este entorno de desarrollo y a su compatibilidad con el resto de herramientas utilizadas, fue utilizado para el desarrollo de SparkEC.

3.2.5 Papyrus

Papyrus es una solución para modelado de sistemas basada en el lenguaje UML (*Unified Modelling Language*) y mantenida por la fundación Eclipse. Se trata de un software gratuito y de código libre, que ha sido desarrollado como un plugin que se ejecuta sobre el IDE Eclipse.

Mediante Papyrus fue posible llevar a cabo un análisis más detallado de la herramienta CloudEC, así como diseñar las diferentes versiones incrementales de la herramienta SparkEC.

3.2.6 Maven

Maven es un sistema de automatización de la construcción de proyectos, así como de gestión de dependencias, desarrollado y mantenido por la fundación Apache.

Maven dispone de un repositorio central de dependencias en el que fue posible encontrar todas aquellas dependencias necesarias para llevar a cabo el desarrollo adecuadamente. Además, es una tecnología bien asentada y muy utilizada en proyectos Java, por lo que fue escogido como el sistema de construcción usado en el TFG.

Capítulo 4

Metodologías de desarrollo

Con este capítulo se describen las diferentes metodologías que se han utilizado para el desarrollo del presente proyecto.

4.1 Scrum

Scrum es una metodología de desarrollo software, desarrollada por Ken Schwaber y Jeff Sutherland, y propuesta por primera vez en la guía de Scrum [17]. Propone un enfoque incremental basado en *timeboxes* para conseguir incrementos en tiempos fijados, y así evitar retrasos. Scrum propone un enfoque en el que todo el equipo se comporta como una unidad, y avanzan juntos hacia un mismo objetivo, permitiendo así que todos los miembros del equipo puedan llevar a cabo todas las tareas, y responsabilizando a todo el equipo Scrum del desarrollo del producto.

4.1.1 Objetivo

Scrum es una metodología encuadrada en las metodologías ágiles que hace un gran énfasis en la gestión de las personas, tratando de controlar los distintos perfiles y roles que puedan surgir, e intentando que todos aporten en el desarrollo.

También pretende minimizar el riesgo en el desarrollo del software, ya que no todo el producto se define al comienzo sino que se va refinando incrementalmente, permitiendo así controlar mejor su desarrollo. A las fases de desarrollo que producen estos diferentes incrementos, cuya duración es fija, se les denomina Sprints.

Es gracias a este desarrollo incremental que es posible obtener un proceso adaptativo que facilite la mejora continua. De este modo, si se determina que existe algún problema que penalice el desarrollo de algún modo, es posible corregirlo y aplicar esta corrección en el siguiente incremento.

4.1.2 Artefactos

En Scrum se definen una serie de artefactos que permiten llevar a cabo la gestión de las tareas. Estos artefactos son los siguientes:

- ***Product Backlog:*** consiste en una lista con toda la funcionalidad pendiente de desarrollar en el sistema. A cada una de las entradas del Product Backlog se la denomina *historia de usuario*, y estas entradas se encuentran priorizadas y estimadas. Esta estimación de las historias de usuario se realiza utilizando los llamados *puntos de historia*, que son puntos con un mismo valor a lo largo del proyecto, pero cuyo valor puede variar en función de los proyectos y organizaciones.

- ***Sprint Backlog:*** conformado por el conjunto de tareas que es necesario cumplir para implementar las funcionalidades seleccionadas para el Sprint actual. Se encuentran estimadas con un mayor detalle que las historias de usuario, y pueden asignarse a los diferentes miembros del equipo.

- ***Sprint Goal:*** este artefacto actúa como una guía para el equipo que le facilita entender por qué se están implementando esas funcionalidades en particular para este Sprint. El Sprint Goal resume todo aquello que se quiere conseguir completando las entradas seleccionadas en el Sprint actual.

4.1.3 El equipo Scrum

En Scrum se propone un equipo en el que todos los miembros cooperan y avanzan unidos hacia un mismo fin. Sin embargo, se definen diferentes perfiles dentro del equipo que tienen distintos cometidos. Estos perfiles son:

- ***Scrum Master:*** se encarga de promover las políticas de Scrum tanto en la organización como en el equipo Scrum. Por ello se requiere que sea un perfil experimentado con la metodología. Además, es importante destacar que el Scrum Master debe actuar al servicio del equipo, y no dirigirlo. De hecho, si se determinara que una persona no funciona adecuadamente como Scrum Master, se podría reemplazar por otro miembro del equipo de desarrollo. No es un rol obligatorio ya que podrían existir equipos Scrum maduros que no necesiten contar con este perfil.

- ***Product Owner:*** es el encargado de asegurar que el Product Backlog se adecúa a las necesidades del cliente y de la organización. Por ello, se puede ver en el Product Owner una doble responsabilidad: por un lado, debe actuar como escudo del equipo ante la organización, ya que los miembros del equipo no deberían atender a peticiones de

otras personas que no sean el Product Owner; y por el otro lado, también es el responsable ante la organización de que el producto desarrollado cumpla con las necesidades existentes.

- ***Equipo de desarrollo:*** lo conforman el conjunto de miembros que desarrollan el producto, manteniendo una responsabilidad conjunta por parte de todo el equipo (no se permite culpar a miembros individuales por problemas en el desarrollo). Por otro lado, el equipo se autoorganiza y son sus propios miembros los que deciden las tareas que hará cada uno. Del mismo modo, las tareas podrían reasignarse entre miembros si fuera necesario para evitar atascos, o simplemente para mejorar la productividad. Se recomienda que el equipo de desarrollo no tenga un tamaño excesivo, ni tampoco muy pequeño. Así, la guía Scrum propone tres miembros como mínimo y nueve como máximo. En caso de que fuera necesario coordinar a equipos más grandes, podría resultar interesante definir un Scrum de Scrums, en el que se establezca una pirámide que permita que un nivel coordine los equipos del nivel inferior.

4.1.4 Eventos

Scrum se enfoca en la gestión de las personas. Por ello, define una serie de eventos cuyo principal objetivo (aunque no único) sería la coordinación del equipo Scrum. Entre los eventos que se encuentran bajo el marco Scrum (ver Figura 4.1) encontramos:

- ***Sprint:*** es la unidad de tiempo fundamental en Scrum. Esta metodología divide el tiempo de desarrollo en Sprints, que actúan como unidades fijas de tiempo en las que se debe entregar la funcionalidad. No es posible retrasar el plazo de entrega de un Sprint, por lo que, en caso de retrasos, simplemente se mueve parte de la funcionalidad al próximo Sprint. Típicamente la duración del Sprint se fija entre dos semanas y un mes, aunque debe ser adaptada a cada equipo.

  Los Sprints deben aportar siempre un valor añadido al producto final. Siempre se selecciona al menos una historia de usuario para desarrollarse a lo largo del Sprint. Sin embargo, en ocasiones se permite introducir un Sprint inicial adicional que no aporte funcionalidad, y que actúe como un Sprint de formación, lo que le permite a un equipo inexperto formarse adecuadamente en la tecnología.

- ***Sprint Planning:*** es una reunión que tiene lugar al principio del Sprint. Su objetivo es seleccionar las entradas del Product Backlog que se desarrollarán en el Sprint, y descomponerlas en tareas técnicas cuyo esfuerzo esté estimado y que sean asignables a miembros del equipo. Así, este evento concluye con la creación del Sprint Backlog para el Sprint actual.

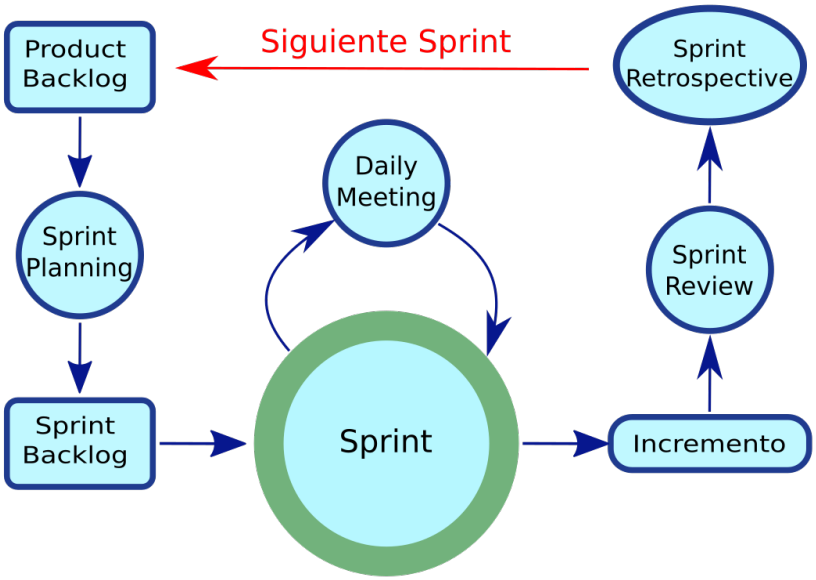

Figura 4.1: Eventos y artefactos en Scrum

- ***Sprint Review:*** reunión que se sitúa al final del Sprint. Se trata de una reunión abierta a la que no solo puede acudir quien lo desee, sino que también se permiten intervenciones de miembros que no formen parte del equipo. El objetivo es mostrar tanto al Product Owner como al cliente la funcionalidad desarrollada. Por ello, en muchas ocasiones adopta la forma de una demo. Por otro lado, no se recomienda preparar excesivamente esta reunión, ni utilizar presentaciones.

- ***Sprint Retrospective:*** este evento consiste en una reunión y se suele encontrar también al final de los Sprints. Tiene un cometido de autoevaluación del equipo Scrum, buscando la mejora continua en la productividad y en el ambiente de trabajo.

- ***Daily Meeting:*** esta reunión tiene lugar diariamente, y puede llevarse a cabo tanto al comienzo del día como al final. Se trata de una reunión informal de sincronización entre los miembros de equipo. Por ello, si bien puede acudir quien quiera, solo se permite que hablen los miembros del equipo Scrum.

  Normalmente, esta reunión es arbitrada por el Scrum Master, que se encarga de que fluya adecuadamente. Se suele hacer de pie y tiene una duración muy reducida (se recomienda que no dure más de 15 minutos). Cada miembro del equipo de desarrollo expone qué ha hecho que aporte valor al equipo desde la última reunión, qué va a hacer ahora, y avisa a los otros miembros si ha encontrado algún obstáculo en el desarrollo.

4.1.5 Aplicación de Scrum

Para el presente TFG se ha utilizado Scrum como la metodología de desarrollo principal. Sin embargo, por las particularidades específicas del proyecto, la metodología no pudo aplicarse de forma literal, requiriendo aplicar una modificación ad hoc.

En particular, es importante tener en cuenta que Scrum hace un gran énfasis en la gestión de las personas que, por las características del TFG donde el equipo de desarrollo consta de una única persona, no aplica en el proyecto.

Por otro lado, se utilizaron todos los artefactos durante el desarrollo. También se llevaron a cabo la mayor parte de eventos de Scrum, como son el Sprint Planning, en el que se definió y estimó una planificación para cada Sprint; la Sprint Review, en la que se evaluaron las mejoras introducidas en cada Sprint; y la Daily Meeting se utilizó para reportar progresos a los directores del trabajo, aunque no se celebrara de forma diaria.

4.2 Kanban

Kanban [18] es una metodología surgida en Japón en las plantas de producción de Toyota como una forma de mejorar la productividad. La palabra Kanban se traduce como *tarjeta de señal*, ya que esta metodología se basa en el uso de tarjetas visuales para llevar a cabo la producción.

4.2.1 Objetivo

El objetivo de Kanban es producir solo aquello que se necesite, en el momento en el que se necesite. A las metodologías que se basan en este principio se las denomina metodologías *Just in Time*. Así, Kanban trata de reducir el desperdicio que se produce en el proceso de producción, llevando a cabo solo las operaciones necesarias.

Además, reduce la multitarea a la que los trabajadores deben ser capaces de atender. Uno de los principios de Kanban, reducir el WIP (*Work in Progress*), consiste precisamente en esto. Se trata de tener el menor número de tareas en curso en cada momento, y no iniciar más tareas si no se finalizan primero las que hay en curso. Se ha comprobado que con esta reducción de la multitarea se consigue una mayor productividad, al aumentar la concentración de los trabajadores.

4.2.2 Tablero Kanban

Para conseguir llevar a cabo los objetivos se propone el uso de tarjetas que se colocarán sobre un tablero (ver Figura 4.2). Este tablero consta de una serie de columnas, que describen las diferentes fases por las que pueden pasar las tareas a realizar. De este modo, las nuevas

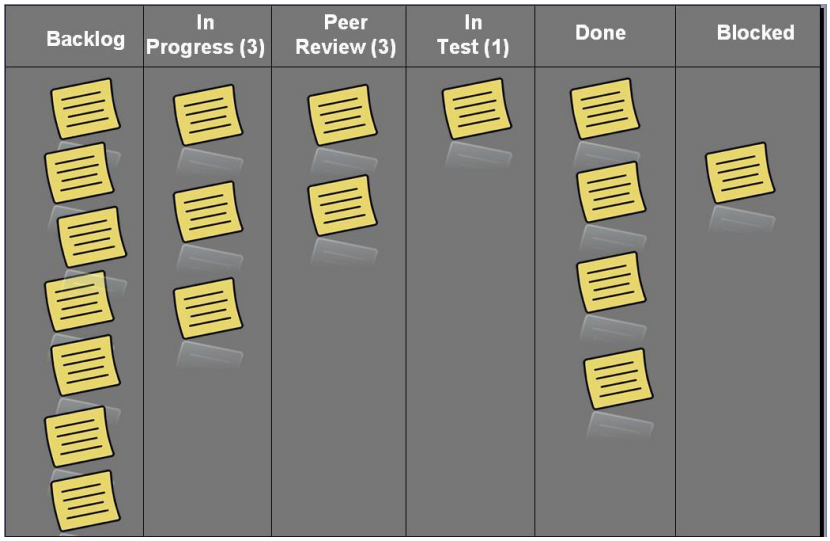

Figura 4.2: Ejemplo de tablero Kanban

tareas se introducen por la parte izquierda del tablero, y fluyen hacia la derecha a lo largo de su tiempo de vida.

El número de columnas que se fijan en el tablero debe adecuarse al dominio o proyecto sobre el que se esté aplicando Kanban. En principio, será necesario establecer al menos una columna con las tareas que actualmente se encuentran en curso, otra para aquellas que hayan sido completadas, y otra para aquellas que estén a la espera; pero es posible modificar esta disposición. Así, si para completar una tarea fuese necesario que esta pasase por varias fases de trabajo, habría que introducir columnas para cada una de esas fases. También pueden introducirse columnas intermedias entre estas fases que actúen como *buffers* de entrada a las fases (contribuyendo así a limitar el WIP).

También es necesario establecer las políticas del tablero que determinan cuándo una tarea puede evolucionar a la siguiente etapa. En ocasiones, cuando el tablero cuenta con un gran número de columnas, estas políticas se colocan de forma explícita sobre el tablero para garantizar que todo el equipo realmente las conozca.

Finalmente, se debe establecer un límite en el número de tareas que se encuentran en progreso. Este límite debe adecuarse a cada equipo, aunque un primer valor que se puede utilizar si no se tiene información acerca del rendimiento del equipo es el doble del número de personas más uno. Gracias a este límite se consigue evitar que exista demasiada multitarea.

4.2.3 Aplicación de Kanban

Actualmente, existen multitud de herramientas informáticas que facilitan la implementación de Kanban. Entre ellas, Trello es una herramienta gratuita que nos permite establecer un

tablero Kanban, crear tarjetas en las columnas del tablero, o mover las tarjetas creadas entre las distintas columnas, entre otras opciones.

En este TFG se ha implementado Kanban mediante Trello estableciendo un tablero que simplemente cuenta con tres columnas: una para aquellas tareas pendientes, otra con las que están en curso, y otra para aquellas que hayan sido completadas.

Mediante el tablero anterior es posible controlar el estado de las tareas que conforman el Sprint Backlog del Sprint actual. Al inicio de cada Sprint, las tareas se añadirán a la columna de tareas pendientes. A lo largo del Sprint, las tareas irán evolucionando por las diferentes columnas hasta que todas ellas alcancen el estado de completadas. En ese momento, se cerrará el Sprint y comenzará el siguiente.

Capítulo 5

Diseño y desarrollo

A continuación, este capítulo introduce las diferentes etapas del desarrollo de SparkEC, intentando reflejar los pasos seguidos en cada uno de los distintos Sprints.

Será posible observar cómo el desarrollo va evolucionando desde los primeros Sprints, con una gran carga en el análisis de CloudEC y de la funcionalidad ofrecida, hasta los últimos, en los que se hará especial énfasis en el rendimiento y la escalabilidad de la aplicación SparkEC, así como en el refinamiento de la arquitectura.

5.1 Primer Sprint: Análisis de CloudEC

Para comenzar, en este primer Sprint se realizó un análisis de la herramienta original, prestando especial atención a su arquitectura, funcionalidad y diseño. A partir de este análisis preliminar se procede con la siguiente fase de reingeniería sobre CloudEC para obtener una nueva herramienta que cumpla con el nuevo nivel de calidad exigido.

Este análisis se realizó en dos etapas diferenciadas. Primero, se hizo sobre la arquitectura de la herramienta apoyado en el lenguaje de modelado C4 [19], que fue diseñado específicamente para utilizarse en este caso; después, se llevó a cabo un análisis a más bajo nivel del diseño de clases que implementan el algoritmo de CloudEC sobre Hadoop.

5.1.1 Planificación

Para acometer este Sprint se plantea una planificación de tareas, que cuentan con su estimación en esfuerzo y coste de acuerdo a la Figura 5.1.

5.1.2 Análisis de la arquitectura

Para llevar a cabo el proyecto de reingeniería propuesto en este TFG primero fue necesario analizar el código fuente de CloudEC mediante ingeniería inversa.

	Nombre de tarea	Trabajo	Nombres de los recursos	Costo
1	**Sprint 1**	**30 horas**		**700,00 €**
2	Análisis de la arquitectura	10 horas	Analista[25%]	300,00 €
3	Estudio del diseño	20 horas	Diseñador[25%]	400,00 €

Figura 5.1: Sprint 1 - Planificación

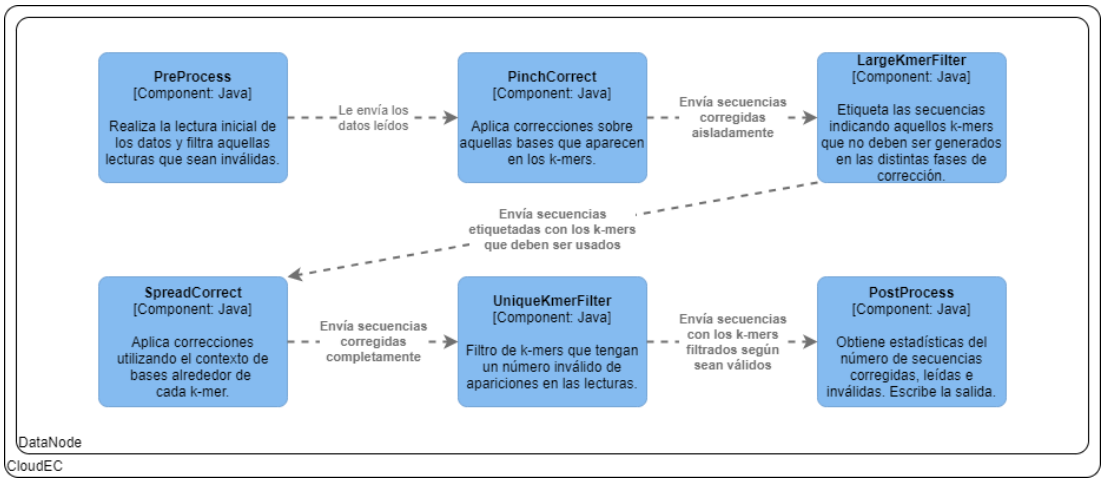

Figura 5.2: Arquitectura de CloudEC

Esta herramienta está diseñada para funcionar de forma distribuida sobre un clúster Hadoop basándose en el patrón arquitectural *Pipe and Filter*. Este patrón se fundamenta en que la información fluye por una serie de fases (denominadas filtros) que progresivamente van transformando los datos hasta obtener la salida deseada.

Para visualizar más fácilmente la arquitectura de CloudEC se ha realizado un diagrama utilizando la notación C4, mostrado en la Figura 5.2. Se pueden observar las seis fases que componen CloudEC y cuyos nombres se conservarán en SparkEC para mantener la relación entre ambas herramientas.

Las funciones principales que cumplen cada una de las fases son:

- ***PreProcess:*** se encarga de realizar la lectura de los datos genómicos de entrada almacenados en HDFS. Además, se llevan a cabo algunas comprobaciones básicas sobre las secuencias, filtrando aquellas que no verifiquen unos mínimos controles de calidad. Por ejemplo, se filtran aquellas secuencias que tienen un gran número de bases desconocidas (bases 'N'), o las que contienen una poliadenilación, esto es, un número elevado de adeninas que pueda deberse a un error en la lectura.

- ***PinchCorrect:*** esta es la primera fase en la que se llevan a cabo correcciones de bases

en CloudEC. Se descompone en dos subfases:

  - *PinchCorrectRecommend*: se alinean los k-mers sin tener en cuenta su base central, es decir, se alinearán aquellos que tengan las mismas bases a ambos lados. Después, se intenta determinar una *base ganadora* para cada grupo de k-mers: aquella que tiene más calidad o mayor número de apariciones que las otras. Tras esto se emiten recomendaciones de modificar las bases que no sigan la base ganadora en aquellos casos en los que la base leída tenga poca calidad.
  - *PinchCorrectDecision*: las recomendaciones emitidas previamente se analizan en conjunto para cada secuencia individual, y se determina si se realizan los cambios sugeridos.

- ***LargeKmerFilter:*** fase en la que no se lleva a cabo ninguna corrección pero que busca etiquetar las secuencias con información útil que permita agilizar el trabajo de corrección posterior. Esta fase busca encontrar k-mers inválidos y etiquetarlos para evitar hacer la alineación sobre ellos. Se descompone nuevamente en dos subfases: *LargeKmerFilterCountKmers* y *LargeKmerFilterTagReads*, donde se genera la información de los k-mers y se etiquetan las secuencias con dicha información, respectivamente.

- ***SpreadCorrect:*** es la segunda fase de corrección que realiza el algoritmo de CloudEC. Si PinchCorrect buscaba hacer correcciones sobre los k-mers, el objetivo de SpreadCorrect es llevar a cabo mejoras sobre las bases que se encuentran alrededor de estos. Así, alinea los k-mers utilizando todas sus bases y mantiene para cada k-mer las bases que se encontraron a su alrededor. De este modo, se pueden alinear las bases alrededor de los k-mers emitidos (igual que se hizo en PinchCorrect con la base central) y emitir recomendaciones de cambios sobre ellas. Tras este proceso, habrá otra subfase de decisión en la que para cada caso se podrán llevar a cabo las modificaciones sobre las bases en función de las recomendaciones recibidas.

- ***UniqueKmerFilter:*** filtro que busca etiquetar aquellos k-mers que tengan un número inadecuado de lecturas, es decir, k-mers que aparecen bien en muy pocas secuencias o bien en demasiadas. Los umbrales son configurables por el usuario. Como sucedía en LargeKmerFilter, en esta fase se deben emitir k-mers, aplicar las comprobaciones correspondientes y etiquetar las secuencias, por lo que también se descompone en dos subfases (*UniqueKmerFilterCountKmers* y *UniqueKmerFilterTagReads*).

- ***PostProcess:*** última fase en la que se generan estadísticas con los resultados de las correcciones realizadas sobre las secuencias de entrada. Además, durante esta fase se escribe la salida final de la ejecución en el formato FASTQ. Para cumplir estos dos objetivos se descompone en dos subfases: 1) *PostProcessMerge*, que mezcla las secuencias

originales con las resultantes del proceso de corrección comparándolas para generar las estadísticas (v.gr. número de secuencias corregidas); y 2) *PostProcessConvert*, que escribe la salida final en HDFS.

Finalmente, se ha observado que CloudEC utiliza formatos de codificación de las secuencias diferentes para la salida y la entrada. Para la salida, utiliza el formato FASTQ mencionado en la Sección 2.1.2; para la entrada, en cambio, utiliza un formato propio llamado SimpleFastQ, similar a FASTQ pero utilizando únicamente una línea por cada lectura. Por lo tanto, es necesario preprocesar los datos de entrada para convertirlos del formato FASTQ al formato SimpleFastQ. Este preprocesamiento penaliza su rendimiento introduciendo una sobrecarga que es proporcional al tamaño del conjunto de datos de entrada, aunque solo es necesario realizarlo una vez para cada conjunto de datos que se desee procesar.

5.1.3 Estudio del diseño de CloudEC

Tras haber hecho un análisis de la arquitectura en fases, es necesario conocer ahora, a más bajo nivel, el diseño de clases que estructuran el funcionamiento general de CloudEC. En este caso se utilizó el lenguaje de modelado UML, en particular, su vista estática con los diagramas de clase.

En la Figura 5.3 se muestra un diagrama de clases en el que es posible observar la estructura general de la herramienta CloudEC. Es importante destacar que, ya que este diagrama se ha extraído a partir del código de un sistema ya desarrollado, se ha intentado que sea lo más verboso posible (incluyendo, por ejemplo, métodos y atributos privados). Esto se ha hecho para poder utilizar los diagramas como un mapa para navegar el código fuente. Este diagrama también permite visualizar las seis fases de CloudEC descritas previamente, descomponiéndose cada una de ellas en dos subfases (a excepción de PreProcess, que solo tiene una).

Además, este diagrama permite detectar puntos de mejora potenciales que se acometerán durante el desarrollo de SparkEC. Por ejemplo, se puede observar la clase *Utils*, en la que se aprecia el antipatrón Blob [20], consistente en que una clase cumple con múltiples responsabilidades y además tiene demasiada complejidad y tamaño. Entre otras cosas, esta clase se encarga de representar a una secuencia, y contiene además toda la lógica necesaria para realizar operaciones sobre las calidades, las bases, e incluso etiquetas para ser usadas por las fases de filtrado. Por otro lado, también se puede ver que la clase que actúa como punto de entrada de la aplicación, *CloudEC*, tiene un método para cada una de las fases, cuyos cuerpos contienen además código repetitivo que puede refactorizarse fácilmente.

Adicionalmente a este primer diagrama de clases que nos ofrece un panorama general, se han desarrollado diagramas que hacen especial énfasis en cada una de las fases por separado, mostrando además las relaciones de herencia y realización de interfaces existentes entre las

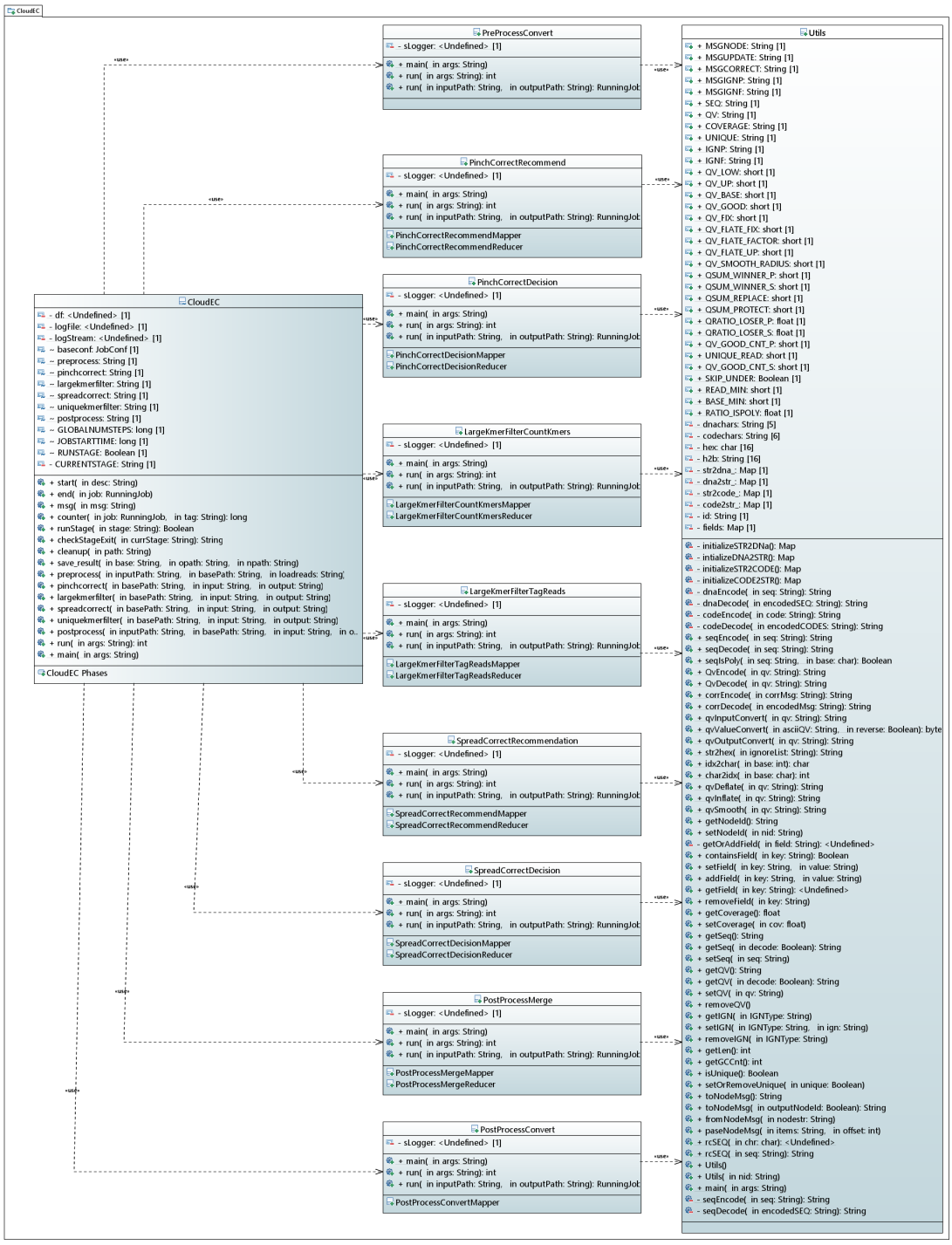

Figura 5.3: Diagrama de clases de CloudEC (vista general)

clases involucradas en esas fases y las clases e interfaces perteneciente a la API de Hadoop. En estos diagramas se muestra una clase que se utiliza en todas las subfases (*Config*) y que

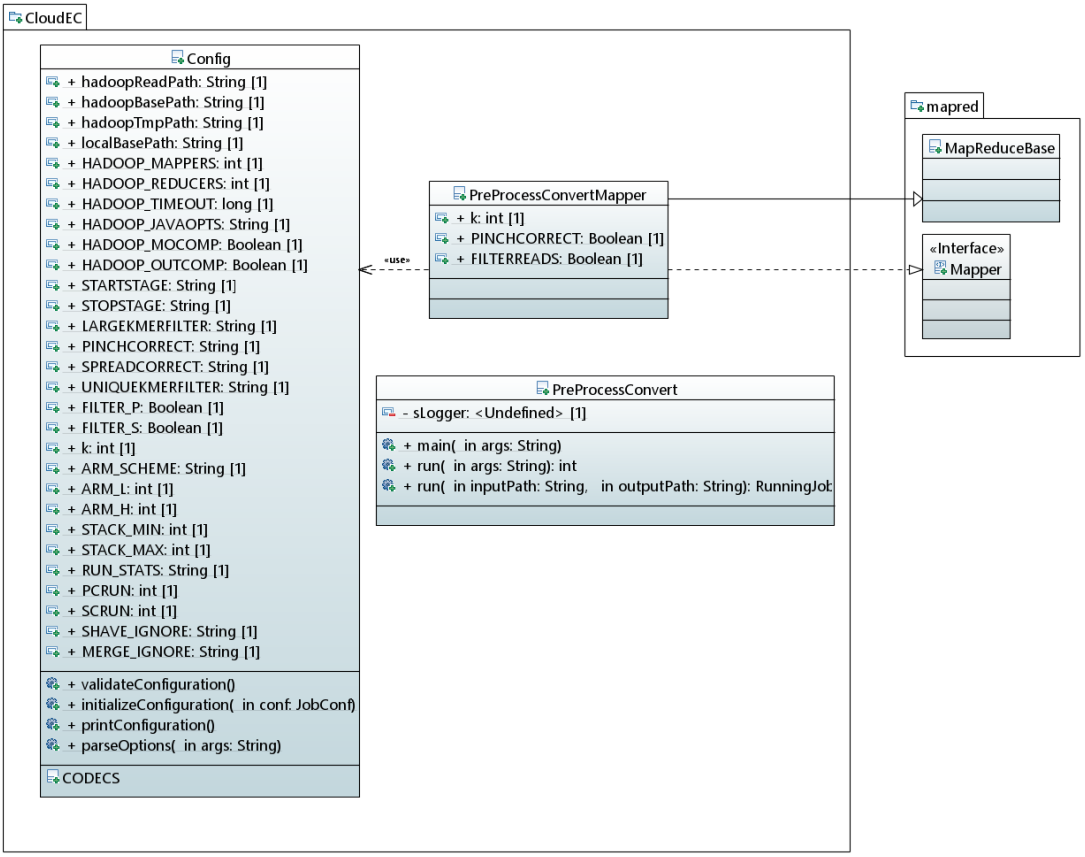

Figura 5.4: Diagrama de clases de CloudEC: PreProcess

contiene información acerca de la configuración con la que se lleva a cabo la ejecución actual. La razón de que se muestre en estos diagramas en vez de en el diagrama general es para mejorar su legibilidad, limitando así los cruces entre líneas. A continuación, se presentan y describen brevemente cada uno de estos diagramas.

PreProcess

La Figura 5.4 muestra el diagrama correspondiente a la clase PreProcess, la cual cuenta con una clase que implementa la interfaz *Mapper*, pero no cuenta con ningún *Reducer*. Esto es debido a que en esta fase solo se hace una comprobación de calidad sobre la entrada, y no es necesario agregar/combinar datos. De este análisis de clases podemos extraer que solo se ejecuta un trabajo MapReduce, que además es muy ligero ya que en él solo se ejecutan tareas *Map*.

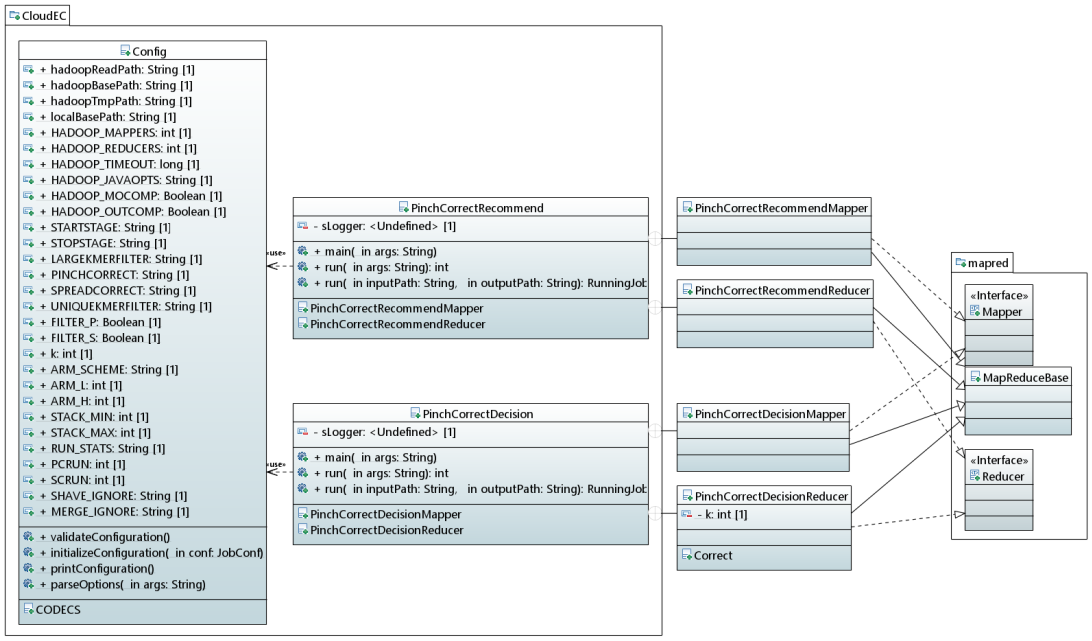

Figura 5.5: Diagrama de clases de CloudEC: PinchCorrect

PinchCorrect

En el diagrama mostrado en la Figura 5.5 podemos ver que existe un mayor número de clases para PinchCorrect. En general, aparecen dos clases que implementan la interfaz *Reducer*, y dos clases que implementan la interfaz *Mapper*. Esto es debido a que PinchCorrect cuenta con dos subfases, siendo capaz cada una de ellas de ejecutar un trabajo MapReduce completo (esto es, *Map*, *Reduce* y *Shuffle and Sort*) por cada intento, ya que PinchCorrect ofrece la posibilidad (mediante configuración) de ejecutarse varias veces.

LargeKmerFilter

En el diagrama para LargeKmerFilter (ver Figura 5.6) también existen dos subfases, constando cada una de ellas de un *Mapper* y un *Reducer*. En este caso sí es posible determinar el número de tareas que se ejecutan (un trabajo MapReduce completo en cada subfase), ya que no se permite que esta subfase se repita varias veces.

SpreadCorrect

En este caso, el diagrama de clases de SpreadCorrect (ver Figura 5.7) se asemeja mucho a PinchCorrect. Esto se debe a que las fases de corrección son más complejas, y hacen uso en ambos casos de clases internas auxiliares para realizar sus operaciones. Además, esta fase también se asemeja a PinchCorrect en el número de trabajos MapReduce que se pueden rea-

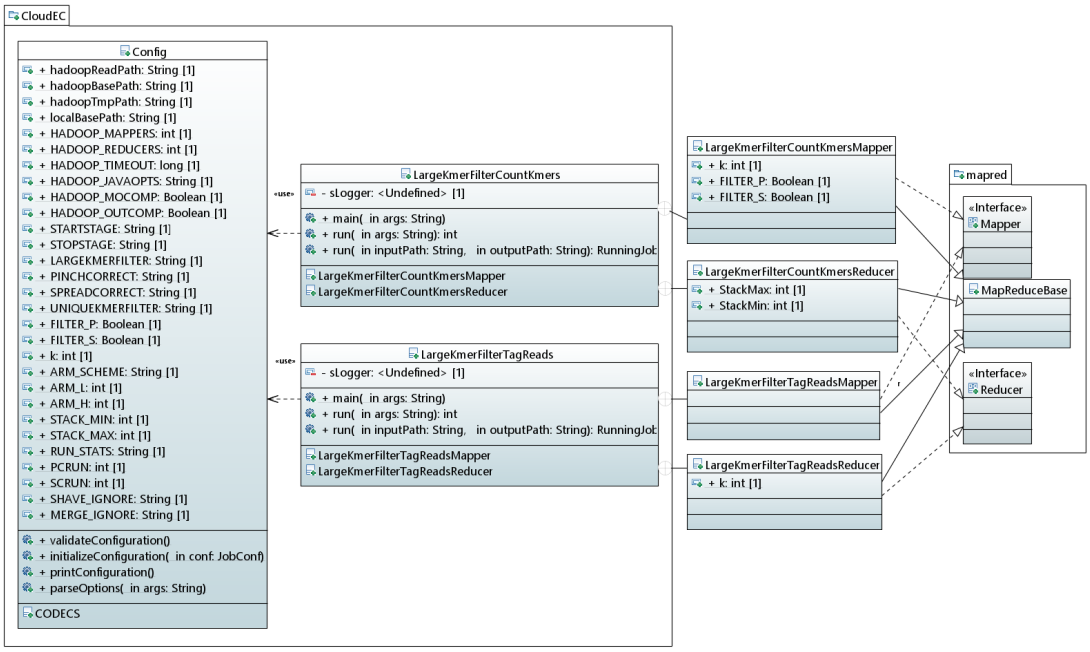

Figura 5.6: Diagrama de clases de CloudEC: LargeKmerFilter

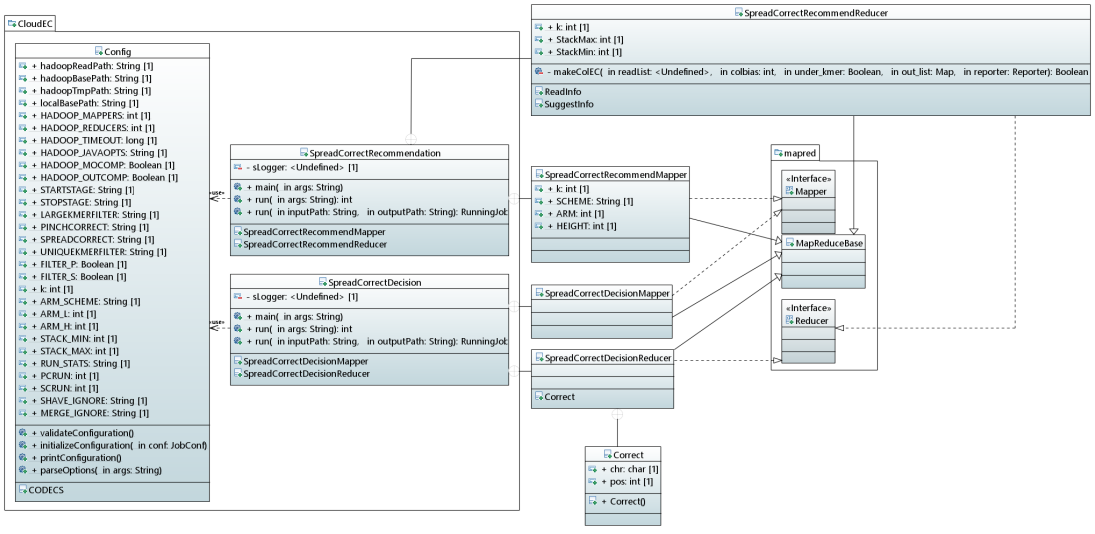

Figura 5.7: Diagrama de clases de CloudEC: SpreadCorrect

lizar en ella, ya que aquí también es posible indicar el número de ejecuciones que se realizan, desencadenando dos operaciones MapReduce completas en cada ejecución.

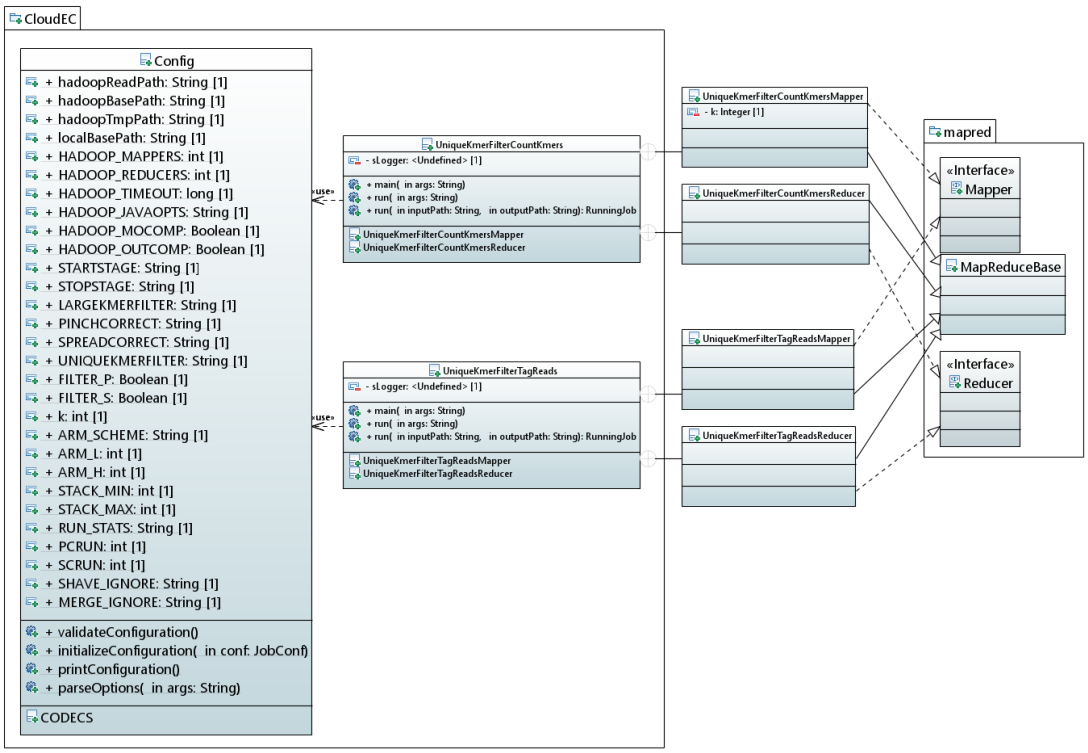

Figura 5.8: Diagrama de clases de CloudEC: UniqueKmerFilter

UniqueKmerFilter

En la Figura 5.8 se puede ver que igual que existía un parecido entre las dos fases de corrección, se encuentran también similitudes entre ambas fases de filtrado. Así, en este caso tampoco existen clases auxiliares, pero sí hay un *Mapper* y un *Reducer* en cada de las subfases en las que se decompone UniqueKmerFilter. También, el número de ejecuciones de este filtro es constante, realizándose siempre dos trabajos MapReduce completos.

PostProcess

Finalmente, en el diagrama correspondiente a PostProcess (Figura 5.9) se ven diferencias respecto a los anteriores. Primero, no hay un *Mapper* y un *Reducer* en cada subfase. Esto es debido a que en la segunda subfase no se hace ningún procesamiento sobre los datos, y simplemente se lleva a cabo la salida del resultado de la ejecución. Por otro lado, se ve también que en esta fase no se hace uso de ninguna clase auxiliar (similar a las fases de filtrado). En este caso, la ejecución siempre implica al menos un trabajo MapReduce parcial (solo con tareas *Map*), y puede también ejecutar otro completo (en caso de que la primera subfase se encuentre habilitada, ya que puede desactivarse en la configuración de la herramienta).

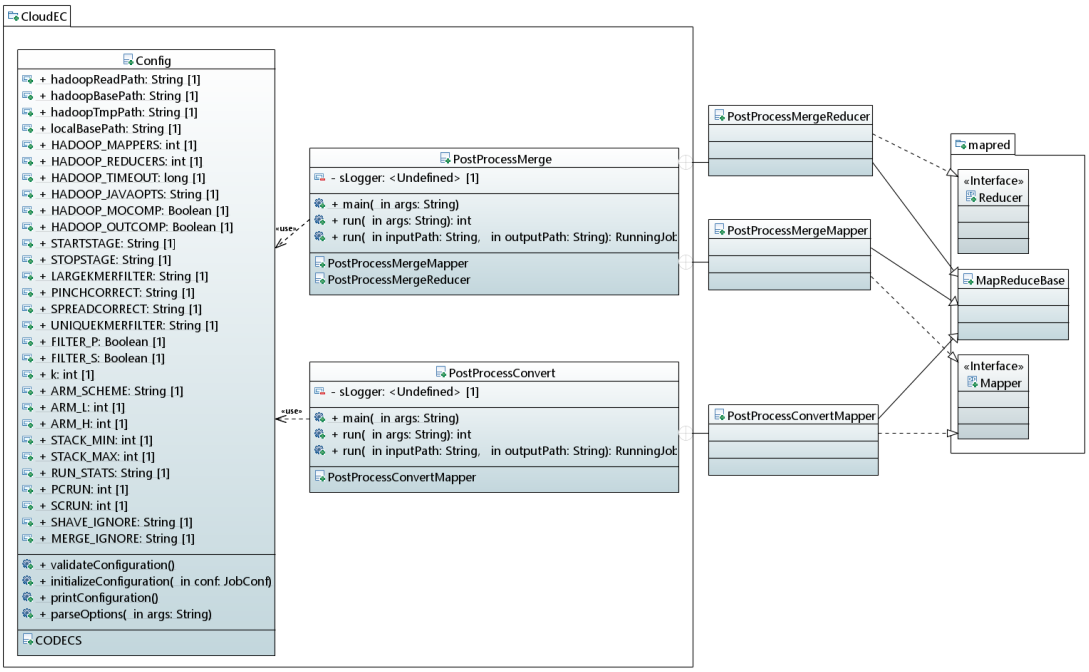

Figura 5.9: Diagrama de clases de CloudEC: PostProcess

5.1.4 Cierre

El resultado de este Sprint ha sido la obtención de todos los diagramas e información acerca del funcionamiento, arquitectura y diseño de CloudEC, lo que nos permitirá obtener la primera versión funcional de SparkEC.

5.2 Segundo Sprint: Implementación directa

Una vez estudiada la herramienta original, y generados los diagramas que nos permiten navegarla sin necesidad de estar consultando continuamente el código, se comienza el segundo Sprint.

El objetivo de este Sprint es obtener una primera implementación lo más directa y fiel posible del algoritmo de corrección de CloudEC sobre el framework Apache Spark. Para ello, se plantea una propuesta de clases y de arquitectura de la herramienta general, conservando la estructura en seis fases definida inicialmente para CloudEC.

5.2.1 Planificación

Para empezar, haremos una planificación del Sprint en la que estableceremos las tareas a realizar, su alcance, duración estimada y recursos. Se propone una primera tarea de diseño

		Nombre de tarea	Trabajo	Nombres de los recursos	Costo
4		**Sprint 2**	**93 horas**		**1.710,00 €**
5		Diseño de alto nivel	5 horas	Analista[25%]	150,00 €
6		Diseño de bajo nivel	10 horas	Diseñador[25%]	200,00 €
7		Implementación	40 horas	Programador[25%]	600,00 €
8		Pruebas	30 horas	Tester[25%]	600,00 €
9		Evaluación de rendimiento	8 horas	Tester[25%]	160,00 €

Figura 5.10: Sprint 2 - Planificación

de alto nivel, en la cual se establece la arquitectura y su estructuración en paquetes. Luego viene una tarea de diseño a bajo nivel para obtener la estructura de clases y métodos en mayor detalle, una de implementación en la que se obtendrá el código de la herramienta, y las pruebas funcionales correspondientes que comprueben que SparkEC se comporta igual fase a fase que CloudEC. Finalmente, una tarea de evaluación preliminar del rendimiento que nos permitirá conocer puntos de mejora para siguientes Sprints. La planificación de este Sprint puede visualizarse en la Figura 5.10.

5.2.2 Propuesta de arquitectura

Una vez completada la planificación del Sprint, comenzamos a desarrollar las funcionalidades. Para empezar, deberemos detallar a grandes rasgos la arquitectura propuesta, con los patrones arquitecturales escogidos.

En general, la arquitectura de SparkEC seguirá el planteamiento de CloudEC. Sin embargo, se estructurará la herramienta en paquetes de clases para cada fase, aislando así las clases.

5.2.3 Diseño UML

Para llevar a cabo la arquitectura propuesta, se ha desarrollado una estructura de clases muy similar a la ofrecida inicialmente por CloudEC. Existe incluso una clase que en esta primera implementación se ha conservado tal cual desde la herramienta original. Esta clase que sobrevive en la primera implementación es *Utils*, que permite representar y operar sobre los nodos y secuencias a lo largo de toda la ejecución. Al reutilizar esta clase, conseguimos una implementación más rápida, que luego pueda ser refinada en iteraciones posteriores.

Se ha desarrollado, igual que se hizo con CloudEC, un diagrama de clases general para SparkEC (ver Figura 5.11), y diagramas de clases específicos para hacer un estudio más pormenorizado de las fases de la herramienta. En este diagrama es posible observar las clases que se han propuesto. Para empezar, encontramos la clase *TimeMonitor*, encargada de llevar

a cabo las mediciones de los tiempos de ejecución de las distintas fases (además del tiempo de ejecución global). También encontramos la clase *Config* que, como su homónima en CloudEC, almacenará toda la configuración de la herramienta.

Por otro lado está la clase *Data*, que actúa como un contenedor, almacenando referencias a aquellos RDDs que necesitemos utilizar en fases posteriores o emitir como salida. Se define también la clase *SparkEC*, que actúa como punto de entrada de la aplicación, y la interfaz *Phase*, la cual deben implementar todas las fases que se incorporen en la herramienta SparkEC.

Por tanto, para llevar a cabo el diseño de cada fase en SparkEC se debe primero definir una clase que implemente la interfaz *Phase*. Además, se pueden introducir clases adicionales para las subfases, o para utilizarlas de forma auxiliar durante su cálculo. Estas fases son, por tanto, muy similares en estructura (como sucedía también en CloudEC), por lo que únicamente se mostrarán a continuación los diagramas de clases de algunas fases a modo representativo.

PreProcess

PreProcess es una de las fases que se han seleccionado como ejemplo para mostrar la estructura de clases. La estructura de PreProcess resulta ligeramente diferente a las demás, ya que se trata de la única fase en la que solo existe una subfase. Además, una mejora planteada para este proyecto consiste en la introducción de la biblioteca HSP descrita en la Sección 3.1.5 (que en este Sprint todavía no se encuentra integrada), por lo que resulta interesante disponer de este diagrama para poder conocer el impacto que tiene ese cambio al llevar a cabo esa integración más adelante (Sección 5.3.3).

En el diagrama mostrado en la Figura 5.12 podemos observar que, por un lado, se ha propuesto la clase *PreProcess*, que implementa la interfaz *Phase* e interactúa con las clases *Data* y *Config* para ejecutar la fase y, por otro lado, la clase *PreProcessTask*, en la que se lleva a cabo la ejecución de la fase finalmente.

PinchCorrect

PinchCorrect es la segunda fase de SparkEC, y la primera de corrección. La estructura de esta fase se analiza para poder contar con un ejemplo de una fase de corrección, ya que como se vio en la estructura de CloudEC, las fases de corrección se parecen entre sí, y lo mismo sucede con las de filtrado.

Como se puede ver en la Figura 5.13, en esta clase ya aparecen dos subfases: *PinchCorrectRecommendation*, que se encarga de realizar el trabajo que CloudEC hace en la clase *PinchCorrectRecommend*; y *PinchCorrectDecision*, que hace el trabajo que CloudEC lleva a cabo en la clase homónima.

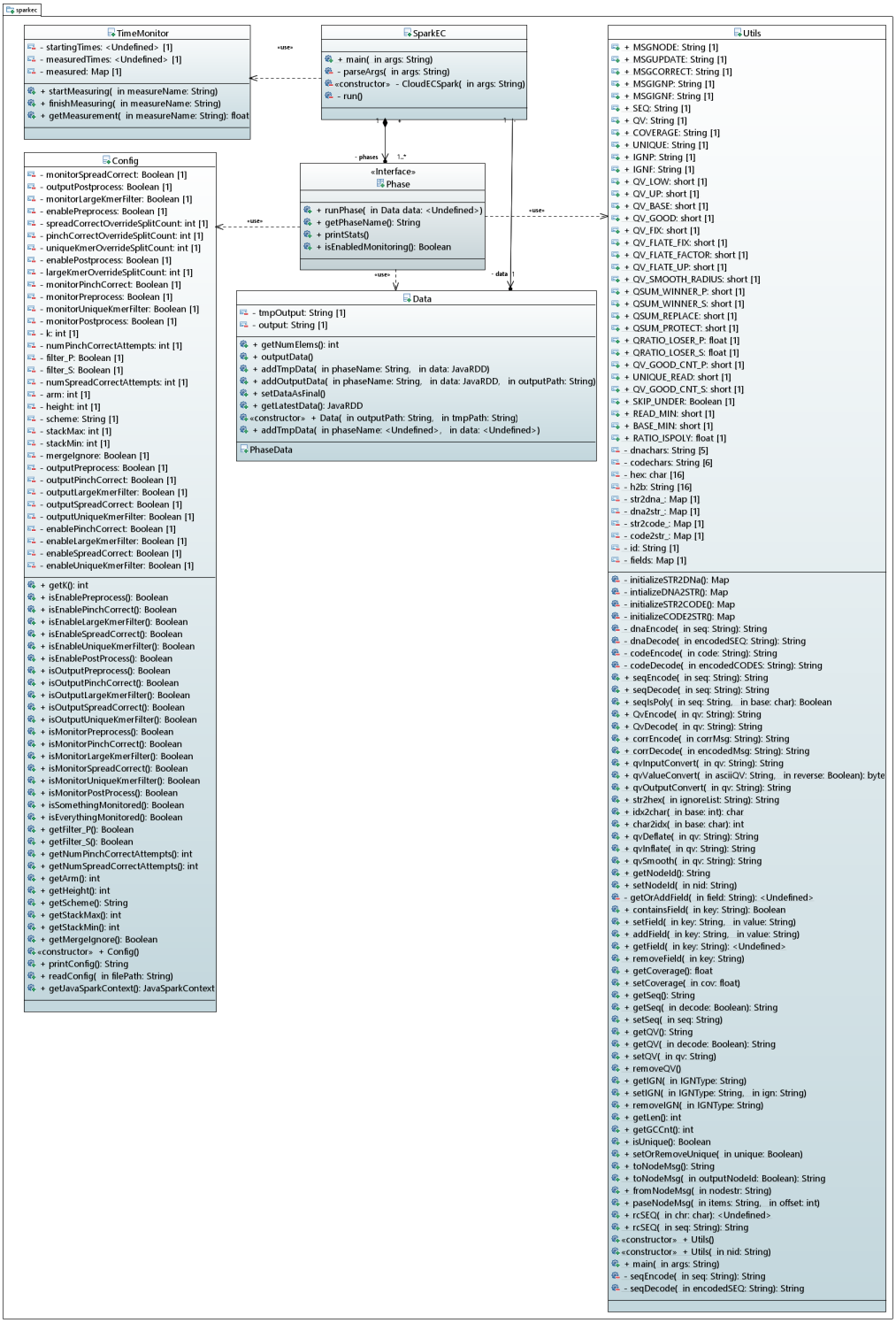

Figura 5.11: Diagrama de clases de SparkEC (Implementación directa)

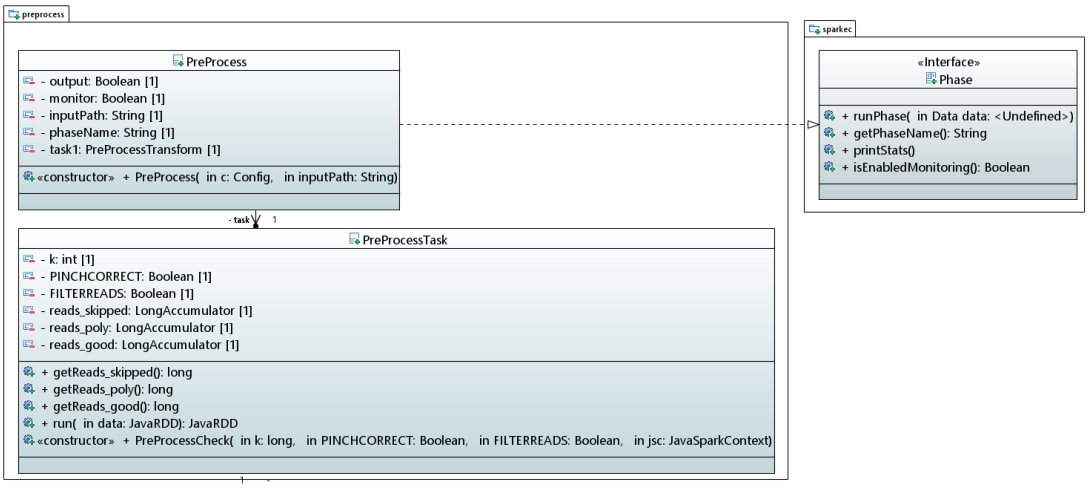

Figura 5.12: Diagrama de clases de SparkEC: PreProcess (Implementación directa)

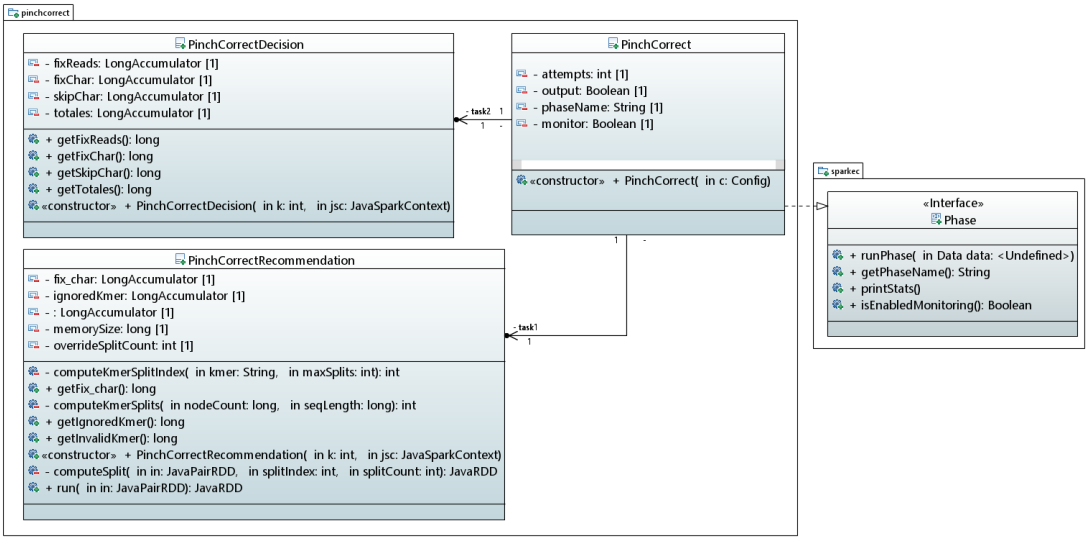

Figura 5.13: Diagrama de clases de SparkEC: PinchCorrect (Implementación directa)

LargeKmerFilter

LargeKmerFilter es la tercera fase de SparkEC, y la primera de filtrado. Igual que antes se ofreció un ejemplo de una fase de corrección, ahora se muestra un ejemplo de fase de filtrado. También, como se hizo con PreProcess, se destaca esta fase porque es sobre la que en un Sprint posterior se va a realizar la denominada optimización de los cortes, que posteriormente también se aplicará sobre las demás fases (Sección 5.3.1).

En este caso, como se ve en la Figura 5.14, las clases que implementan las subfases son *CountKmers*, que hace el trabajo de la clase *LargeKmerFilterCountKmers* de CloudEC, y *Tag-*

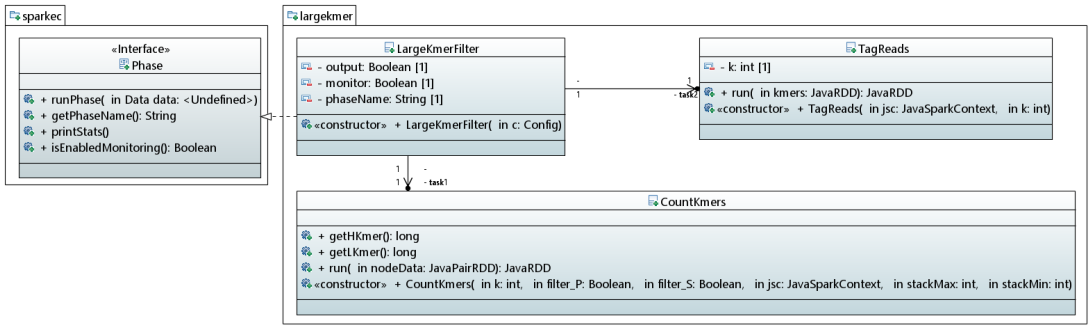

Figura 5.14: Diagrama de clases de SparkEC: LargeKmerFilter (Implementación directa)

Reads, que es el equivalente a *LargeKmerFilterTagReads* en CloudEC.

5.2.4 Implementación

Una vez que contamos con una propuesta de las clases necesarias para obtener esta primera versión de la herramienta SparkEC, debemos abordar su implementación. Para ello, se hará una conversión de las distintas operaciones MapReduce de CloudEC en transformaciones y acciones a realizar sobre los RDDs de Spark.

Por otro lado, como el objetivo de este Sprint es obtener una versión sobre Spark con el mismo comportamiento que su equivalente Hadoop, el proceso de validación se realizará comparando la salida obtenida utilizando conjuntos de datos de entrada conocidos de esta versión y de CloudEC. Es esperable que ambas herramientas produzcan una salida similar, si no igual, en las diferentes subfases, por lo que las pruebas de SparkEC que se implementarán aprovecharán esta ventaja para poder hacer la validación.

5.2.5 Cierre

Este segundo Sprint finaliza con una primera versión de SparkEC que ya es capaz de ejecutar el algoritmo de corrección completo. A partir de este punto, se construirán sobre esta versión distintas optimizaciones y mejoras que permitan obtener finalmente una herramienta lo suficientemente eficiente y escalable.

5.3 Tercer Sprint: Primeras optimizaciones

Tras obtener una primera versión funcional de SparkEC, se comienzan a barajar posibles optimizaciones y puntos de mejora que puedan llevarse a cabo sobre la herramienta. En un primer momento se encuentran varios posibles cuellos de botella que deben analizarse y mejorarse a lo largo de este Sprint.

5.3.1 Planteamiento de optimizaciones

En primer lugar, las fases centrales del algoritmo en las que se llevan a cabo operaciones de filtrado o de corrección generan un gran volumen de datos intermedios, que en muchas ocasiones deben transmitirse por la red y/o persistirse en disco. Normalmente, estos datos consisten en k-mers anotados con información de las secuencias en donde se encontraron. Por ello, el primer punto de mejora consiste en limitar el número de k-mers que se emiten durante estas fases, consiguiendo de este modo tener operaciones de *Shuffle and Sort* que no sean tan pesadas. Para resolver este problema se propone descomponer el algoritmo en varias etapas o cortes, generando así una parte de los k-mers en cada etapa, en vez de generarlos todos de una vez y tener que operar con ellos simultáneamente.

Por otro lado, la información de las lecturas en los k-mers se utiliza para poder unirlos posteriormente con las lecturas que los originaron. Esto provoca directamente un shuffle que involucra a un alto volumen de datos (todos los k-mers generados y todas las secuencias de entrada simultáneamente). Para resolver este problema, se propone almacenar la información de las lecturas en cada k-mer. De este modo, no habrá que volver a unir los k-mers con las lecturas, y bastará con simplemente operar sobre los k-mers.

También se descubre que la información de las lecturas se usa repetidamente en cada una de las fases. La tercera optimización que se propone consiste en persistir, utilizando los mecanismos que nos proporciona Spark para ello (como se explica en la Sección 3.1.4), las secuencias que vayan a utilizarse en múltiples ocasiones.

Finalmente, se propone la integración en SparkEC de la biblioteca HSP mencionada en la Sección 3.1.5, mediante la cual es posible realizar la lectura de los datos de entrada directamente en formato FASTQ, en vez de realizar el preprocesado de los mismos que hace CloudEC (y, hasta este punto, también SparkEC), como se mencionó en la Sección 5.1.2. Eliminar esta necesidad de hacer el preprocesado de los datos reduce el tiempo de procesamiento global, por lo que esta mejora cobra especial interés. Además, esta mejora aporta comodidad al usuario final, ya que en CloudEC era responsabilidad suya realizar el preprocesado de forma externa a la herramienta.

5.3.2 Planificación

Para llevar a cabo todos los cambios y mejoras que se proponen se han planteado las tareas que conforman el Sprint Backlog y que se muestran en la Figura 5.15.

5.3.3 Diseño y desarrollo

En primer lugar se debe modificar el comportamiento de aquellas fases que tengan un mayor impacto en la cantidad de k-mers generados, introduciendo el procesamiento del conjunto

	Nombre de tarea	Trabajo	Nombres de los recursos	Costo
10	**Sprint 3**	**35 horas**		**740,00 €**
11	Análisis de optimizaciones	10 horas	Analista[50%]	300,00 €
12	Diseño de la integración con HSP	3 horas	Diseñador[50%]	60,00 €
13	Implementación de la integración con HSP	2 horas	Programador[17%]	30,00 €
14	Diseño de los cortes	2 horas	Diseñador[50%]	40,00 €
15	Implementación de los cortes	8 horas	Programador[33%]	120,00 €
16	Implementación de la persistencia	1 hora	Programador[13%]	15,00 €
17	Modificación de representación de kmers	1 hora	Programador[13%]	15,00 €
18	Evaluación de rendimiento de las optimizaciones	8 horas	Tester[50%]	160,00 €

Figura 5.15: Sprint 3 - Planificación

de datos en distintas etapas o cortes. Para ello, al tratarse de una primera versión experimental, se introduce un cambio en la fase LargeKmerFilter, en particular en la clase *CountKmers*. En el diagrama de la Figura 5.16 se puede observar la inclusión de tres nuevos métodos en esta clase: 1) *computeKmerSplits*, que se encarga de calcular el número de etapas (o cortes) que se harán sobre la entrada; 2) *computeSplit*, que permite realizar el cálculo de una única etapa; y 3) *computeKmerSplitIndex*, mediante el cual es posible determinar el corte concreto en el que se debe emitir cada uno de los k-mers. En la Figura 5.17 se muestra un diagrama de secuencia en el que es posible comprobar el funcionamiento del sistema de cortes descrito.

También, para llevar a cabo la optimización consistente en almacenar información de las secuencias en los k-mers no es necesario modificar la estructura de las clases, ya que la comunicación entre fases se realiza codificando los valores sobre Strings, y la lógica de la ejecución no se ha visto alterada. Bastará, por tanto, con cambiar la implementación de las diferentes subfases.

Lo mismo sucede con la persistencia de los RDDs que contienen las lecturas. La introducción de esta optimización no provoca cambios en la estructura de clases de la herramienta, por lo que únicamente se ha modificado la implementación de la clase *Data*, en la que se almacenan las referencias de estos RDDs.

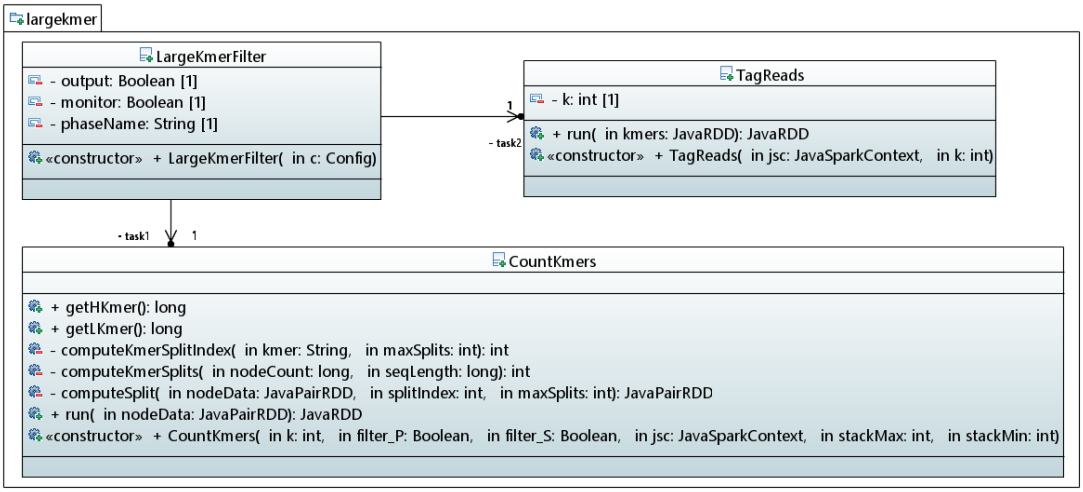

Figura 5.16: Diagrama de clases de LargeKmerFilter - Introducción de los cortes

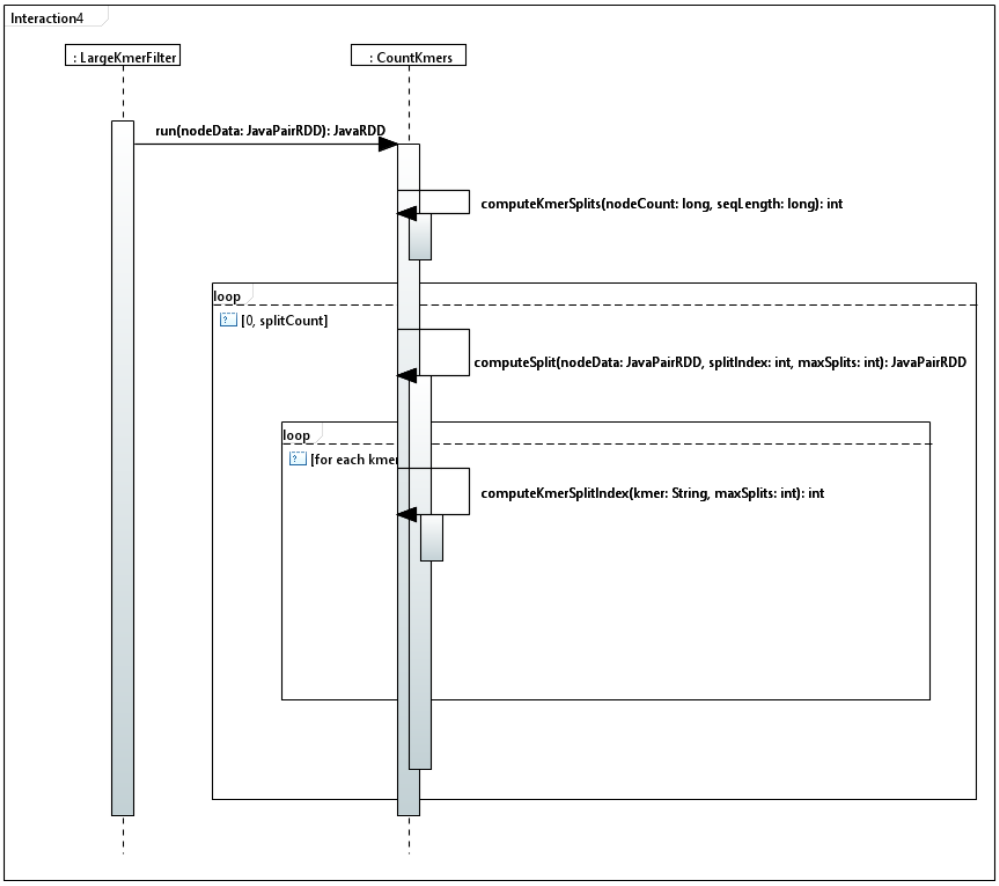

Figura 5.17: Diagrama de secuencia de LargeKmerFilter - Introducción de los cortes

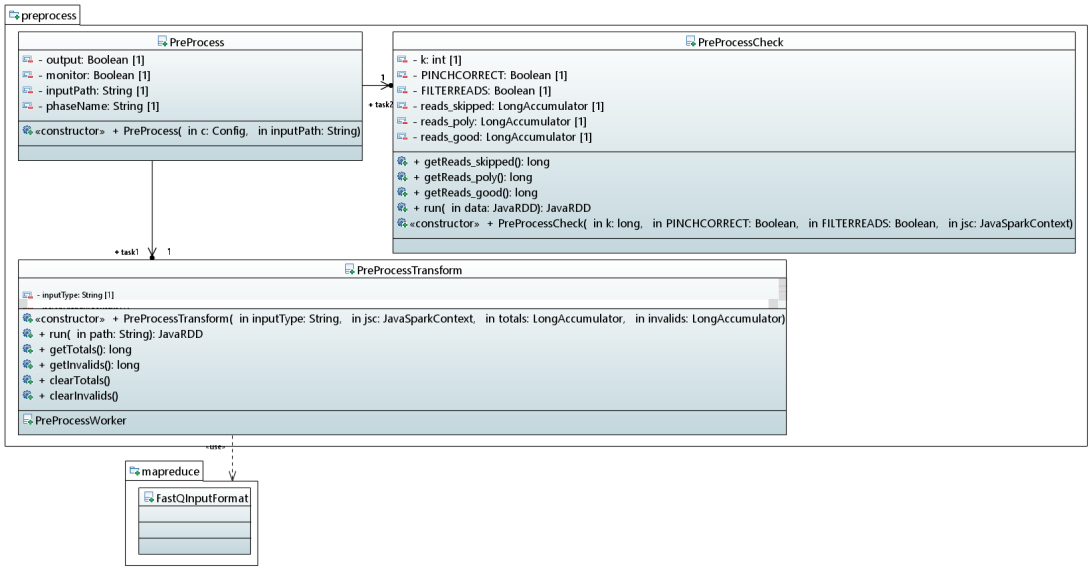

Figura 5.18: Diagrama de clases de PreProcess - Integración con HSP

Por otro lado, se debe modificar el diseño de clases correspondiente a la fase PreProcess para permitir la integración de HSP para la lectura de datos de entrada. Para ello, se subdivide PreProcess en dos subfases: 1) PreProcessTransform, que se encarga de realizar la lectura de la entrada; y 2) PreProcessCheck, que realiza el control de calidad de las secuencias que previamente realizaba la clase *PreProcessTask* (ver Figura 5.12). De este modo, la comunicación con HSP se realiza a través de la nueva subfase PreProcessTransform, limitando así el acoplamiento entre ambos sistemas a una única clase (ver Figura 5.18).

Finalmente, con respecto a las pruebas, si bien no fue necesario en este Sprint incluir nuevas pruebas (bastó con comprobar que efectivamente se superaban las existentes), sí que se modificaron algunos de los tests. Al alterar la representación de los datos utilizada en las subfases, fue necesario introducir en las pruebas operaciones de preprocesado para obtener formatos comparables con CloudEC, por lo que también hubo que hacer trabajo de codificación en este aspecto.

5.3.4 Cierre

Tras implementar las distintas optimizaciones, se realizaron una serie de pruebas del rendimiento obtenido con cada una de ellas para determinar si finalmente proporcionaban una mejora real, o si por el contrario era necesario revertirlas. A continuación, se muestran las conclusiones alcanzadas:

- **Se conserva la optimización de los cortes.** Esto es, la realización de una división de los k-mers generados en varias etapas. Esta optimización produjo una reducción en

el uso de memoria, y la herramienta se volvió más estable siendo capaz de procesar conjuntos de datos de mayor tamaño.

- **Se revierte la anotación de los k-mers con datos de lecturas.** Si bien esta optimización podía evitar algunas operaciones, anotar los k-mers con información extra producía un incremento en el uso de memoria que penalizaba el tiempo de ejecución de la herramienta.
- **Se conserva la persistencia de lecturas.** En este caso se comprobó que mantener estos datos persistidos impactaba positivamente en el tiempo de ejecución, por lo que este cambio no se eliminó.
- **Se conserva la integración con HSP.** En este caso hubo una mejora clara: se eliminó la necesidad de preprocesar los datos de entrada y el tiempo requerido para ello, para pasar a realizar la lectura directamente desde los ficheros FASTQ.

5.4 Cuarto Sprint: Optimizaciones de escalabilidad

En este Sprint se abordaron algunas inestabilidades de la herramienta cuando se sometía a situaciones en las que el conjunto de datos tenía un gran tamaño para la memoria disponible. Por ello, todas las optimizaciones que se proponen en este Sprint están enfocadas a reducir el consumo de memoria de SparkEC para conseguir mejorar su capacidad para procesar grandes conjuntos de datos y, por tanto, incrementar su escalabilidad.

5.4.1 Planteamiento de optimizaciones

En primer lugar, los RDDs que se persistían explícitamente se mantenían en memoria hasta el final de la ejecución cuando, como máximo, iban a utilizarse en la fase siguiente. Por ello, se plantea mejorar la gestión de la persistencia de los RDDs, evitando mantenerlos más tiempo del necesario en memoria, y liberando así espacio para otros propósitos.

Además, la comunicación entre las distintas subfases, e incluso entre operaciones de la misma subfase, se realiza de forma ineficiente. Todos los campos (secuencias, calidades, identificadores, valores temporales…) se codifican sobre objetos String, lo que incrementa el uso de memoria ya que Java representa internamente cada carácter con dos bytes (usa codificación UTF-16). Esta codificación sobre Strings se hace por dos razones diferentes en CloudEC: en primer lugar, porque la propia representación de las lecturas y secuencias en la herramienta se hace con Strings, y en segundo lugar porque los mensajes intermedios entre subfases se transmiten directamente sobre Strings utilizando únicamente caracteres de tabulación como separadores (en unos casos realizando esta codificación con métodos estáticos definidos en la clase *Utils*, y en otros casos simplemente concatenando los campos en la salida de las subfases).

	Nombre de tarea	Trabajo	Nombres de los recursos	Costo
19	**Sprint 4**	**78 horas**		**1.410,00 €**
20	Análisis de las optimizaciones	8 horas	Analista[50%]	240,00 €
21	Diseño de las nuevas clases	16 horas	Diseñador[50%]	320,00 €
22	Implementación de las nuevas clases	32 horas	Programador[37%]	480,00 €
23	Implementación de la memoria compartida	12 horas	Programador[13%]	180,00 €
24	Cacheado de RDDs	2 horas	Programador[50%]	30,00 €
25	Evaluación de rendimiento de las optimizaciones	8 horas	Tester[50%]	160,00 €

Figura 5.19: Sprint 4 - Planificación

Para la primera situación se propone crear una estructura de clases, en la que sea posible contar con diferentes implementaciones de secuencias y lecturas que puedan ser más eficientes. Se proponen además varias implementaciones: la primera de ellas se enfoca en permitir compartir memoria entre las secuencias y los k-mers que se generen de ellas; la segunda en almacenar cada secuencia consumiendo la menor cantidad de memoria posible, pero sin permitir compartir memoria.

Para el segundo problema se presenta la posibilidad de desarrollar clases ad hoc para la comunicación entre las diferentes subfases. Mediante el uso de estas clases debería ser posible evitar la sobrecarga en el uso de memoria debida a la utilización de Strings para la transmisión de información.

Finalmente, al utilizar clases para representar esta información, puede resultar interesante utilizar la librería de serialización Kryo [21], que permite reducir el consumo de memoria y el tráfico de red durante las operaciones de shuffle, ya que serializa los objetos de forma más compacta y eficiente que el mecanismo de serialización por defecto de Java. Esta es una opción de configuración que proporciona Spark, no pudiendo utilizarse por tanto en CloudEC (al menos no de una forma transparente).

5.4.2 Planificación

En base al análisis de las optimizaciones que se ha hecho previamente, se han planteado una serie de tareas que permiten implementar y evaluar toda la funcionalidad de este Sprint y que se muestran en la Figura 5.19.

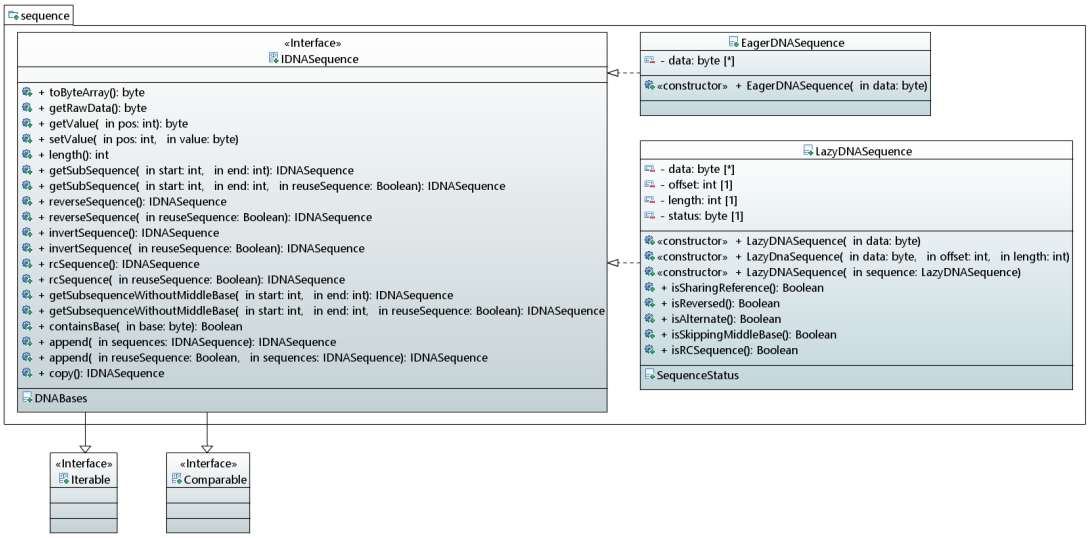

Figura 5.20: Diagrama de clases de SparkEC - Secuencias

5.4.3 Diseño y desarrollo

Para la primera optimización consistente en evitar que se mantuvieran persistidos RDDs innecesarios hubo que modificar la clase *Data*, para que solo mantuviera cacheado el último conjunto de datos generado y evitar así desperdiciar memoria con datos antiguos. Este cambio, por tanto, no provocó realmente una alteración en la estructura de SparkEC (igual que había sucedido ya con la introducción de la persistencia de RDDs).

En el caso de la segunda optimización (evitar el uso de Strings en la comunicación entre fases y subfases), debemos plantear un nuevo diseño en el que descompongamos la clase *Utils*, que actualmente realiza muchas tareas diferentes, en distintas clases para cada una de ellas. Se ha identificado que *Utils* se encarga de representar secuencias de bases, lecturas completas, y de toda la codificación y decodificación de las bases y calidades en diferentes formatos que permitan aplicar las correcciones u obtener una representación comprimida de la información. Para cada una de esas responsabilidades se propondrá, al menos, una clase.

En primer lugar, para la representación de secuencias se propone una nueva jerarquía de clases e interfaces. Para permitir su visualización, se ha desarrollado un diagrama de clases que se muestra en la Figura 5.20. Se puede observar que se ha definido una nueva interfaz, *IDNASequence*, en la que se detallan las operaciones que todas las clases que representen secuencias deben tener disponibles. Entre ellas podemos destacar las siguientes operaciones:

1. ***reverseSequence:*** gracias a esta operación es posible revertir el orden de las bases (la primera base sería la última, la segunda sería la penúltima y así sucesivamente).

2. ***invertSequence:*** operación que permite modificar todas las bases de la secuencia por

su complementaria (esto es, adeninas se cambian por timinas, timinas por adeninas, citosinas por guaninas, guaninas por citosinas, y el resto se dejan igual).

3. ***rcSequence:*** este método realiza simultáneamente las operaciones previas (i.e. *invertSequence* y *reverseSequence*).

Cabe destacar que además se han definido versiones alternativas de cada una de estas operaciones en las que se recibe un argumento adicional: *reuseSequence*. Este argumento booleano permite que se reutilice el propio objeto secuencia al llamar a una operación, reduciendo así el número de instancias creadas, y limitando de esta manera el impacto en el rendimiento que puede tener el recolector de basura de la máquina virtual de Java. Los métodos que no admiten este argumento se comportan de forma equivalente a aquellos que sí lo admiten y reciben un valor 'falso' como entrada para ese argumento.

Para esta nueva interfaz *IDNASequence* se proponen dos implementaciones distintas:

- ***EagerDNASequence:*** se trata de la más directa ya que utiliza un array de bytes para almacenar y representar cada una de las bases. En caso de que se realice alguna operación sobre la secuencia, se reservará otro array de bytes que almacene las nuevas bases.

- ***LazyDNASequence:*** la segunda implementación trata de aplicar la optimización propuesta para este Sprint de utilizar memoria compartida entre las secuencias y los k-mers generados. En este caso, al realizar operaciones sobre la secuencia, simplemente se modifican los campos *status*, *offset* y *length*, conservando la referencia al array de bytes previo. De este modo no se instancia un nuevo array al llevar a cabo las operaciones, y la memoria se comparte entre la secuencia original y la nueva mientras no se ejecute la recolección de basura sobre ambos objetos. El nombre de esta segunda implementación (*Lazy*) hace referencia al hecho de que las bases no se modifican en el momento de ejecutar la operación, sino cuando se solicita el valor de la base sobre la nueva secuencia.

Por otro lado, se ha definido también la clase *Node* que representa una lectura. Por ello, también dispone de campos asociados con información sobre las calidades de las bases, así como metadatos generados por las lecturas y operaciones de filtrado que pueda haber. Esta clase contiene la secuencia de bases definida previamente. El resultado de estos cambios se puede apreciar en el diagrama de clases mostrado en la Figura 5.21.

La introducción de la librería de serialización Kryo implicó modificar el código de inicialización, tanto de la herramienta como de las fases, para permitirles registrar en Kryo tanto las clases temporales que ya existían como las nuevas que se introdujeron.

Además, la clase *Utils* sigue conservando la capacidad de cumplir las responsabilidades descritas al principio de esta sección, a pesar de que existen clases que permitan realizar es-

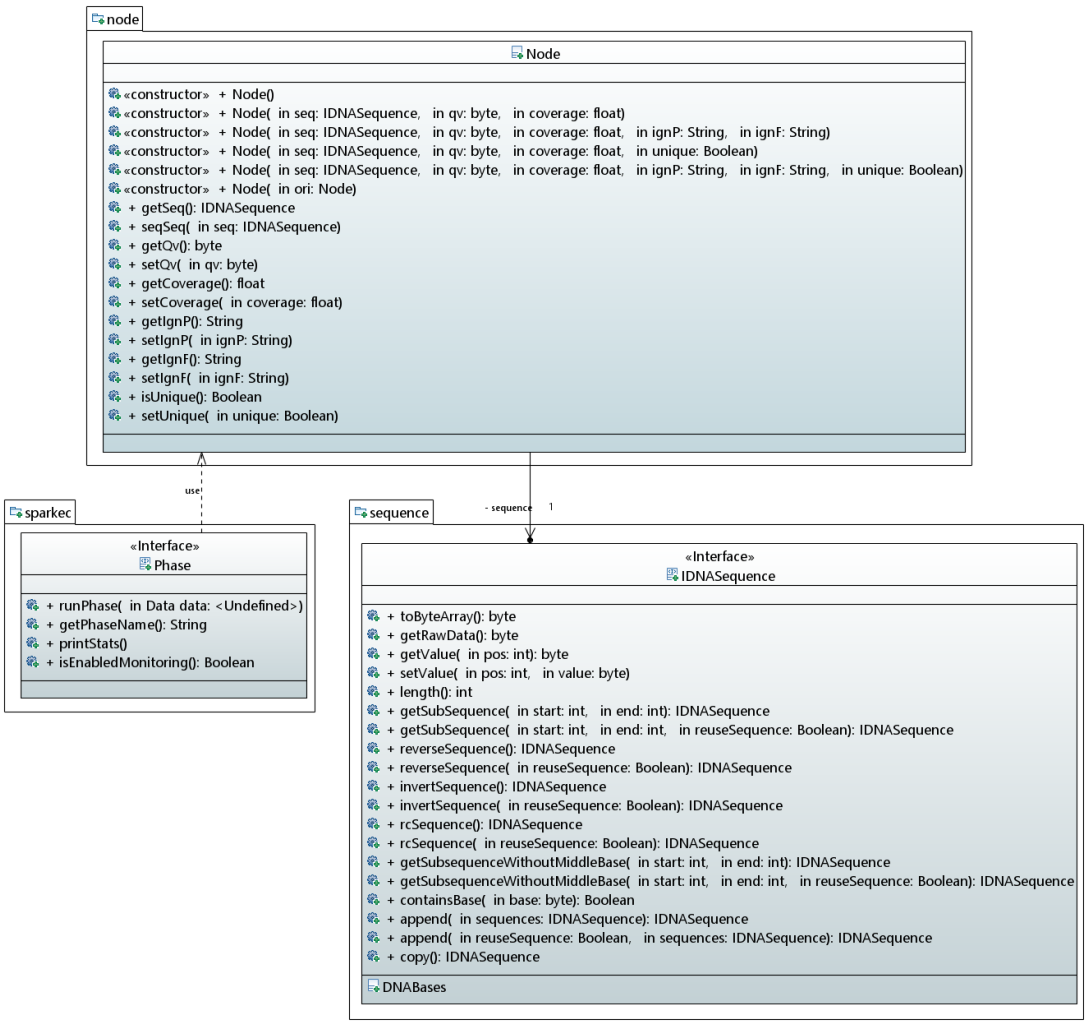

Figura 5.21: Diagrama de clases de SparkEC - Nueva clase Node

tas tareas de forma más específica. La razón de no eliminar durante este Sprint estas operaciones ahora innecesarias es que así podemos desplazar la actualización del código de las pruebas al siguiente incremento, ya que no sería necesario modificarlo para comprobar que efectivamente las fases funcionan, y podrían seguir necesitando la clase *Utils* para facilitar la interoperación entre los formatos actuales y los usados por CloudEC.

Finalmente, en las fases se modificaron muchas transformaciones para que, en vez de emitir datos codificados sobre un String, se hicieran sobre clases ad hoc, capaces de almacenar esos datos de forma más eficiente.

5.4.4 Cierre

Tras la implementación y evaluación de las optimizaciones realizadas, se han tomado las siguientes decisiones:

- **Se conserva la limpieza de datos que ya no deben permanecer persistidos**. Esto elimina una sobreutilización innecesaria de la memoria, por lo que se ha decidido seguir adelante con esta optimización.
- **Se conserva la nueva representación de las secuencias**, ya que ha demostrado ser más eficiente. La implementación que se fija por defecto es *EagerDNASequence*, ya que se comprobó experimentalmente que esta implementación utilizaba menos memoria. Esto puede deberse a que la implementación *LazyDNASequence* forzaba a replicar la secuencia completa en los nodos, en vez de solo los k-mers, y además introducía una sobrecarga al almacenar los campos *offset*, *length* y *status*.
- **Se conserva la nueva representación de las lecturas.** Esta nueva representación ha resultado ser menos costosa en memoria que su equivalente disponible en *Utils* (que almacenaba la información codificándola sobre un String).
- **Se conserva la introducción de la librería Kryo.** El uso de esta librería redujo, con muy poco esfuerzo de desarrollo, el uso de memoria total por parte de la aplicación.
- **Se conserva el uso de clases ad hoc para la comunicación entre subfases.** El uso de estas clases, en conjunto con la librería Kryo, consigue reducir satisfactoriamente la memoria consumida, por lo que también se mantuvo este cambio.

5.5 Quinto Sprint: Optimizaciones de estructuras de datos

Con este último Sprint, y una vez resueltos los problemas que penalizaban gravemente el rendimiento, se intentará obtener una versión final con estructuras de datos optimizadas, además de con una configuración simplificada respecto a los Sprints anteriores. Adicionalmente, se realiza la evaluación del rendimiento para las optimizaciones consideradas.

5.5.1 Planteamiento de optimizaciones

En primer lugar se ha observado que aplicar la optimización de los cortes del tercer Sprint sobre todas las fases ha tenido un impacto positivo. Pero surge un problema, y es que el valor de los cortes utilizado para cada fase no se está estimando adecuadamente, lo que obliga a utilizar valores prefijados manualmente para poder obtener una mejora significativa en los tiempos. Es por ello que se propone una mejora en cómo se realiza la estimación de los cortes.

	Nombre de tarea	Trabajo	Nombres de los recursos	Costo
26	**Sprint 5**	**35 horas**		**645,00 €**
27	Análisis de las optimizaciones	4 horas	Analista[50%]	120,00 €
28	Diseño de nuevo sistema de cortes	4 horas	Diseñador[50%]	80,00 €
29	Implementación de nuevo sistema de cortes	13 horas	Programador[50%]	195,00 €
30	Introducción de particionadores específicos	6 horas	Programador[50%]	90,00 €
31	Evaluación de rendimiento	8 horas	Tester[50%]	160,00 €

Figura 5.22: Sprint 5 - Planificación

También se ha determinado que el número de particiones de los RDDs tiene un gran impacto en el rendimiento, por lo que en este Sprint se investigará cuáles son las mejores estrategias de repartición, y cuál es el número de particiones ideal para cada caso, intentando obtener si es posible un mecanismo que lo calcule automáticamente.

Finalmente, y aunque no influya en el desempeño de la herramienta, se aplicarán las actualizaciones pendientes sobre el código de pruebas del Sprint anterior en este incremento, permitiendo así reducir finalmente la funcionalidad de la clase *Utils*.

5.5.2 Planificación

Para completar este último Sprint, y siguiendo la estructura establecida en Sprints anteriores, se ha definido un conjunto de tareas a realizar y que se muestran en la Figura 5.22.

Entre ellas se encuentran un análisis de las optimizaciones a realizar, el diseño de un nuevo sistema de cortes, su implementación, la introducción de particionadores, y la evaluación de rendimiento.

5.5.3 Diseño y desarrollo

En primer lugar se ha decidido implementar mejoras en el diseño de la solución de los cortes. Existen operaciones similares que se hacen en varias fases, y operaciones específicas de cada fase, por lo que se ha tomado la decisión de desacoplar el comportamiento específico de los cortes que no depende de las fases en otro paquete. Para implementar esta aproximación, se propone el uso del patrón de diseño estrategia, ofreciendo así una interfaz con las funciones que debe ofertar cada estrategia de cortes, y una implementación por defecto. Este rediseño

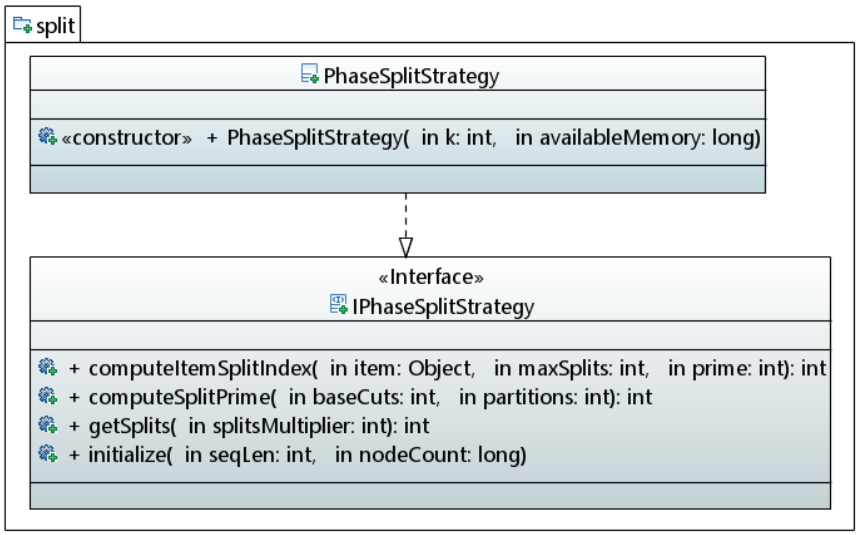

Figura 5.23: Diagrama de clases de Split - Rediseño de cortes

se puede observar en el diagrama de clases mostrado en la Figura 5.23.

En el diagrama es posible observar que se hace referencia al cálculo y uso de un número primo. Esto se debe a que podrían aplicarse tanto diferentes estrategias de particionado en Spark, como distintas implementaciones de la estrategia de cortes. Por tanto, podría darse un escenario en el que tanto el particionado de Spark como la división de los cortes se realizasen empleando el mismo algoritmo para determinar la distribución de los datos. En este caso, podrían crearse tareas Spark que no realizasen ningún trabajo ya que el algoritmo que distribuye los datos en cada corte podría determinar que todos los elementos procesados por esta tarea no deben ser tratados todavía en el corte en el que se encuentra. Por tanto, la distribución de las tareas entre los nodos no sería eficaz. La introducción del número primo (que no necesariamente debe ser primo, sino que basta con que sea coprimo con el número de particiones utilizado) permite evitar este problema, ya que consigue que no haya divisores comunes entre el número de particiones y el número de cortes a utilizar, logrando efectivamente que no se generen tareas Spark vacías independientemente de las implementaciones concretas de las estrategias utilizadas. También hay que destacar que este mecanismo no garantiza que el volumen de datos a procesar se distribuya de forma homogénea en los distintos cortes. Para tratar de paliar este problema, la implementación ofrecida por defecto buscará un número primo que sea mayor o igual que el número de cortes utilizado, con lo que en el peor caso podría existir algún corte que como máximo procese hasta el doble de datos que otro. El Listado 5.1 muestra el código que determina en qué corte debe procesarse cada k-mer en la implementación por defecto, basada en el uso del método *hashCode*.

Por otro lado, hubo que modificar la estrategia de particionado de Spark. Esta, por defec-

```
public int computeItemSplitIndex(Object item, int maxSplits, int
    prime) {
  int aux = item.hashCode();
  if (aux < 0) {
    aux *= -1;
  }
  return (aux % prime) % maxSplits;
}
```

Listado 5.1: Código de distribución de los k-mers en los cortes

to, fijaba un valor muy bajo de particiones tras llevar a cabo una operación de shuffle. Este comportamiento resultaba especialmente nocivo para el algoritmo utilizado por SparkEC ya que este genera un gran volumen de datos temporales y, en Spark, existe un límite máximo de 2 GB para el tamaño de cada partición (a partir del cual, la ejecución de la tarea falla). Para resolver este problema se optó por utilizar, de forma constante a lo largo de toda la ejecución, tanto el mismo número de particiones como la misma estrategia de particionado de los datos. Este número de particiones es dependiente del tamaño de la entrada y es, generalmente, mayor que el número de particiones por defecto al realizar un shuffle (aunque tanto este número como el número de particiones al leer un fichero son dependientes de la configuración de Spark utilizada). Por lo tanto, en la práctica este cambio implicó elevar el número de particiones utilizado.

Finalmente, se reducirá la clase *Utils*. Para ello se conservarán aquellos métodos estáticos que permiten codificar y decodificar cadenas (para conseguir formatos comprimidos y compatibles con CloudEC), pero se eliminarán todo el resto de funcionalidades que existen en esta clase ya que, como hemos visto, se han implementado en otras clases y paquetes. Debido a este cambio, se ha decidido renombrar la clase como *EncodingUtils*, y se ha movido al paquete *node*. Estos cambios en el diseño se ven reflejados en la Figura 5.24. Para poder reducir la clase *Utils* a la actual *EncodingUtils* se actualizó también el código de las pruebas para no utilizar estas clases más que para su responsabilidad actual de codificación y decodificación de cadenas.

5.5.4 Cierre

Tras el diseño, implementación y evaluación de estos últimos cambios, se presentan las decisiones tomadas para cada una de las optimizaciones:

- **Se conserva el nuevo sistema de cortes.** Este sistema, aparte de tener un diseño más limpio que el anterior, tiene ventajas en rendimiento ya que es capaz de estimar adecuadamente un valor más próximo al ideal de cortes para cada ejecución.
- **Se conservan los cambios hechos sobre los particionadores.** Elevar el número de

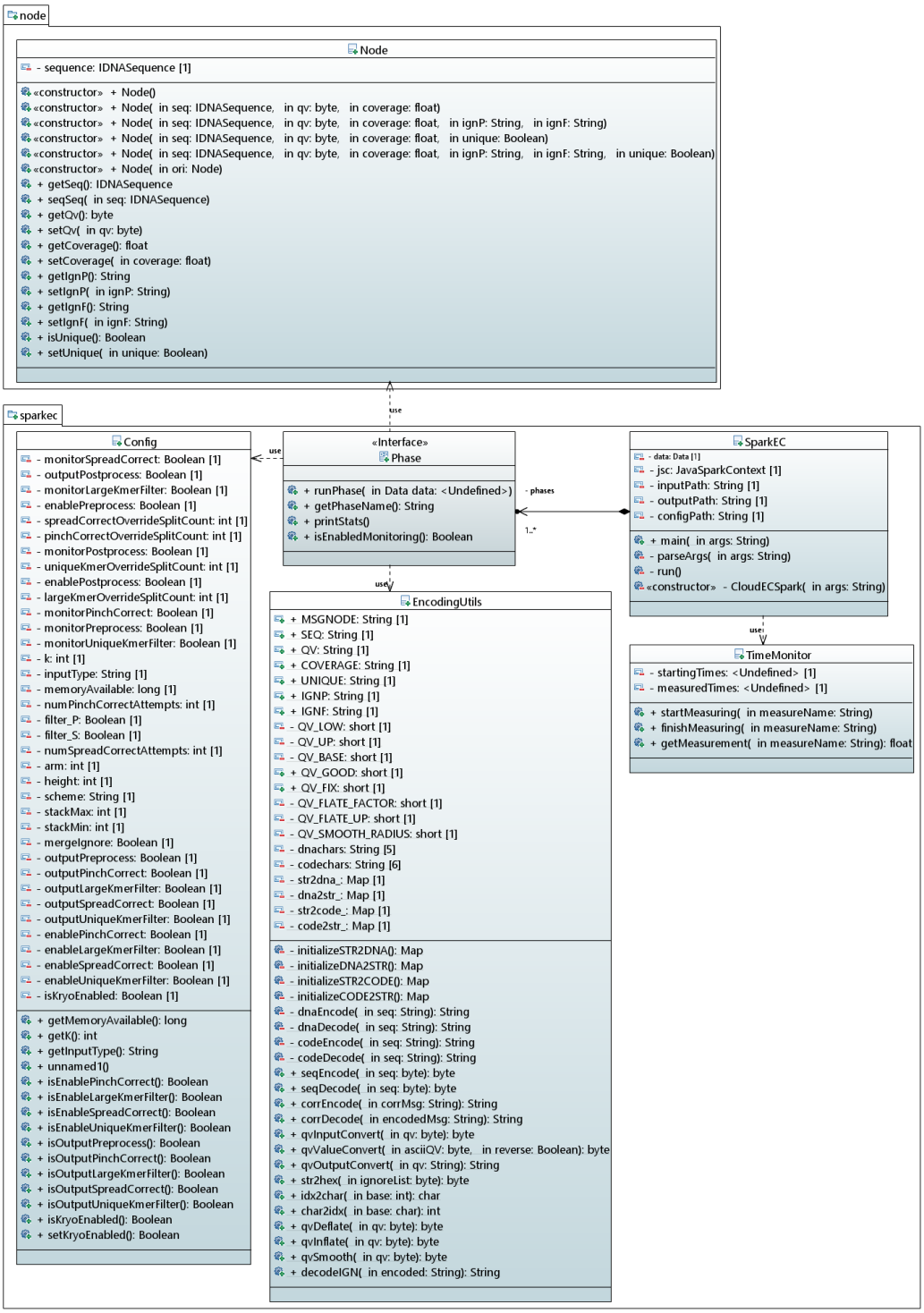

Figura 5.24: Diagrama de clases de SparkEC - Utils reducido

particiones, si bien penalizó el tiempo de ejecución de la herramienta, también incrementó la estabilidad de la misma, conservándose de este modo el cambio introducido.

- **Se mantiene el cambio de *Utils* por *EncodingUtils*.** Este cambio no se esperaba que tuviera ningún impacto en el rendimiento, pero sí mejoró el diseño de la herramienta, por lo que se mantuvo.

5.6 Estimaciones y coste

La herramienta se desarrolló a lo largo de los cinco Sprints previamente descritos. Es importante destacar que para el desarrollo no fue posible contar, desde un primer momento, con una planificación completa de inicio a fin del proyecto. En parte esto fue debido a que se empleó la metodología Scrum, basada en el refinamiento progresivo del Product Backlog en base a los resultados obtenidos; y en parte a que no era posible conocer de antemano las optimizaciones que sería necesario aplicar para alcanzar una versión competitiva. Por ello se ofrecieron las planificaciones que se acometieron en cada Sprint al inicio del mismo, que sí pudieron extraerse como parte de su Sprint Backlog. Sin embargo, puede ser interesante de cara a conocer los costes del proyecto disponer de un resumen de todos los Sprints, que se muestra en la Tabla 5.1. Además, se incluye un diagrama de Gantt en la Figura 5.25 para aportar un enfoque más visual de todo el desarrollo.

Para llevar a cabo la estimación de costes se ha intentado tener en cuenta los diferentes perfiles que intervendrían en las tareas de desarrollo en un proyecto real (aunque en el TFG, prácticamente todos ellos son asumidos por el alumno, con excepción del perfil del Director), asignándoles a cada uno de ellos los costes mostrados en la Tabla 5.2.

Además, en el proyecto también hay que destacar otras tareas que fue necesario acometer a mayores del propio desarrollo, como la evaluación experimental mostrada en el Capítulo 6 para conocer el rendimiento de la herramienta de forma comparativa con CloudEC, el estudio inicial en el campo del Big Data y de la Bioinformática, la redacción del presente documento, o las revisiones realizadas por los directores. En la Tabla 5.3 se puede apreciar el esfuerzo y los costes en cada apartado.

Respecto a los costes de las tareas diferentes al desarrollo de la herramienta, tanto para el estudio del entorno como para la redacción de la memoria se ha estimado un coste de 15€/hora; por otro lado, para la evaluación experimental, se han asumido 20€/hora en el esfuerzo del alumno.

Nombre	Comienzo	Fin	Esfuerzo	Coste
Sprint 1	14/10/2019	01/11/2019	30 hh	700€
Sprint 2	04/11/2019	17/12/2019	93 hh	1.710€
Sprint 3	03/02/2020	13/02/2020	35 hh	740€
Sprint 4	13/02/2020	11/03/2020	78 hh	1.410€
Sprint 5	11/03/2020	25/03/2020	35 hh	645€
		Total	271 hh	5.205€

Tabla 5.1: Planificación global del desarrollo (hh=horas-hombre)

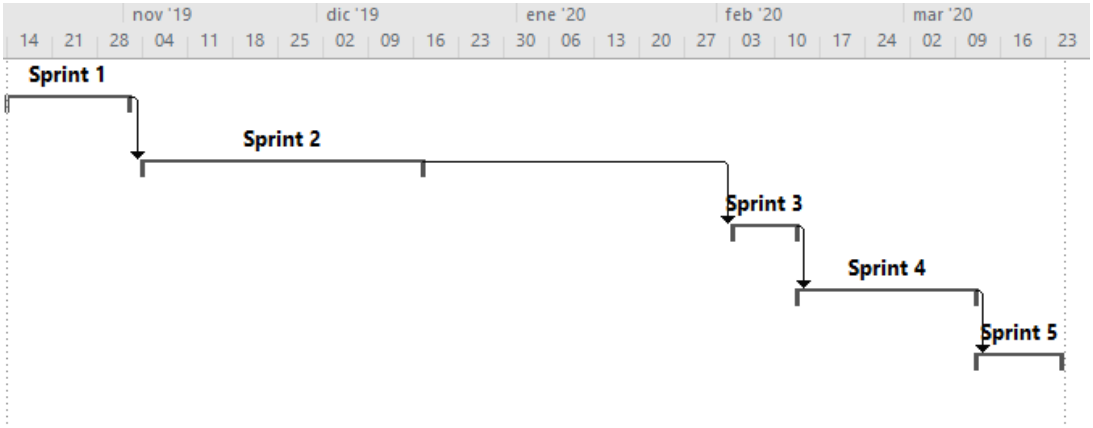

Figura 5.25: Diagrama de Gantt del proyecto

Perfil	Coste/hora
Director	50€/h
Analista	30€/h
Diseñador	20€/h
Programador	15€/h
Tester	20€/h

Tabla 5.2: Coste por perfil

Ámbito	Trabajo	Coste
Estudio del entorno	30 hh	450€
Desarrollo de la herramienta	271 hh	5.205€
Evaluación del rendimiento	75 hh	1.500€
Redacción de la memoria	70 hh	1.050€
Revisión de los directores	35 hh	1.750€
Total	481 hh	9.955€

Tabla 5.3: Coste y esfuerzo globales (hh=horas-hombre)

Capítulo 6

Evaluación experimental

Tras finalizar el último Sprint de desarrollo de SparkEC, se llevó a cabo una evaluación del rendimiento en un entorno clúster de forma comparativa con la herramienta tomada como punto de partida (CloudEC). Para este análisis se tuvieron en cuenta tanto los tiempos de ejecución globales como fase a fase.

A continuación, se presentan los conjuntos de datos utilizados en la evaluación, la configuración del entorno de pruebas y el análisis de los resultados obtenidos.

6.1 Conjuntos de datos

Para llevar a cabo esta evaluación se escogieron cuatro conjuntos de datos en formato FASTQ con diferentes características en cuanto a número de secuencias y longitud (número de bases) por secuencia (ver Tabla 6.1). Tres de ellos (D1, D2 y D3) ya se habían utilizado en el estudio original del rendimiento de CloudEC [1]. Estos conjuntos de datos son públicos y pueden encontrarse mediante su identificador correspondiente en la base de datos de secuencias del NCBI (National Center for Biotechnology Information) [22, 23].

6.2 Configuración del entorno

El entorno de pruebas utilizado fue Pluton [24], un clúster para computación de altas prestaciones. Este clúster dispone de un nodo frontend accesible desde Internet mediante ssh y desde el cual es posible solicitar los recursos computacionales proporcionados por los nodos de cómputo y lanzar trabajos paralelos para su ejecución (ver Figura 6.1).

Respecto a la configuración software del clúster, Pluton hace uso de la suite Rocks basada en GNU/Linux CentOS 7 (v7.7.1908). Entre las diferentes versiones de la máquina virtual de Java disponibles en el clúster, se hizo uso de la versión OpenJDK 1.8.0_242. Además, Pluton utiliza el gestor de recursos SLURM mediante el cual es posible solicitar múltiples nodos de

Nombre	Identificador	Número de secuencias	Longitud de secuencia
D1	SRR352384	26.030.862	152
D2	SRR022866	12.775.858	152
D3	SRR034509	10.353.618	202
D4	SRR4291508	25.232.347	100

Tabla 6.1: Conjuntos de datos utilizados

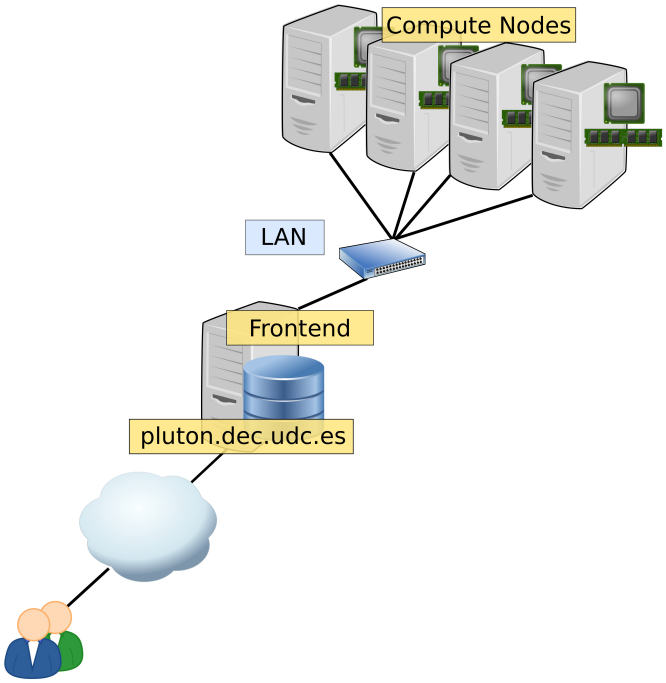

Figura 6.1: Esquema de la estructura general del clúster Pluton

cómputo necesarios para las pruebas de evaluación. Como se mencionó en la Sección 3.1.6, el uso de la herramienta BDEv permite el despliegue de frameworks Big Data de manera sencilla y se integra con el gestor de recursos del clúster de forma transparente para el usuario.

Respecto al entorno hardware, Pluton consta de 20 nodos de cómputo que se encuentran divididos en dos particiones o cabinas. En las pruebas de este TFG se hizo uso de la primera partición, pues es la que dispone de un mayor número de nodos (18 en total). La Tabla 6.2 muestra detalles sobre la configuración hardware de los nodos utilizados. Respecto a la configuración específica de los frameworks Hadoop y Spark, en cada caso se adaptó para conseguir los mejores resultados posibles en cada uno de los escenarios que se presentan a continuación.

Modelo de CPU	2 x Intel Xeon E5-2660 Sandy Bridge EP
Frecuencia de reloj de CPU	2.20 GHz
Frecuencia de reloj Turbo de CPU	3 GHz
Núcleos por CPU	8
Threads por núcleo	2
Caché L1	32 KB
Caché L2	256 KB
Caché L3	20 MB
Memoria RAM	64 GB DDR3 1600 MHz
Disco duro	HDD 1 TB SATA3 7.2K rpm
Interfaces de red disponibles	InfiniBand FDR & Gigabit Ethernet

Tabla 6.2: Especificaciones hardware de los nodos utilizados

6.3 Análisis de los resultados

Se presentan a continuación los resultados obtenidos para cada uno de los cuatro conjuntos de datos usando dos valores diferentes de K (i.e. la longitud de los k-mers) y variando el número de nodos desde 5 hasta 13. Es importante tener en cuenta en el análisis de los resultados que, tanto en el caso de CloudEC como de SparkEC, siempre hay un nodo que no realiza cómputo, debido a la arquitectura líder-trabajador de los frameworks Hadoop y Spark.

- ***D1.*** Los resultados obtenidos para D1 demuestran que SparkEC es aproximadamente 3 veces más rápido de media que CloudEC para todos los escenarios y valores de K (ver Figura 6.2). Además, esta mejora asciende hasta 3,4x cuando se utilizan 13 nodos, lo que demuestra que su escalabilidad es superior.

- ***D2.*** Los datos mostrados en la Figura 6.3 permiten observar también una mejora en todos los casos, consiguiendo SparkEC tiempos significativamente menores. En este escenario el volumen de datos es más reducido que para D1 debido al menor número de secuencias de entrada (ver Tabla 6.1), lo que beneficia a SparkEC, permitiéndole manejar, sin necesidad de recurrir a elevar el número de cortes, todos los datos en memoria. Por ello, las mejoras obtenidas son superiores, alcanzando una aceleración máxima de 3,7x cuando se usan 13 nodos y $K = 55$.

- ***D3.*** De acuerdo a la Tabla 6.1 este conjunto presenta un número de secuencias similar a D2. Sin embargo, la longitud de secuencia es superior (202 vs 152), lo que impacta directamente en el número de k-mers generados y, por tanto, en la memoria necesaria para ellos. Como mencionamos anteriormente, un uso de memoria muy elevado puede limitar la mejora potencial que SparkEC puede proporcionar respecto a CloudEC. Sin

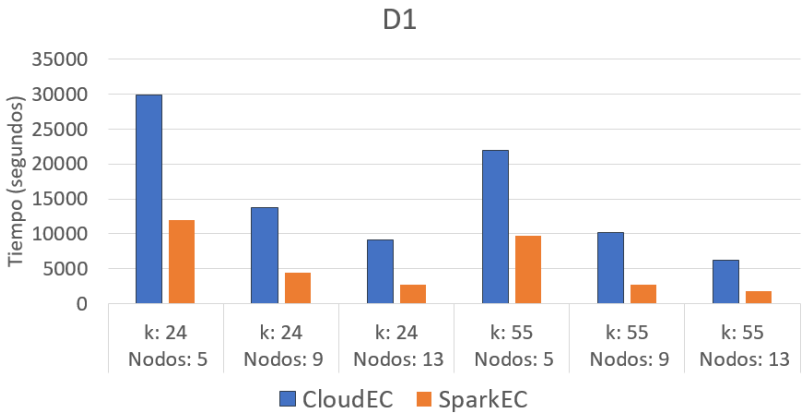

Figura 6.2: Resultados para D1

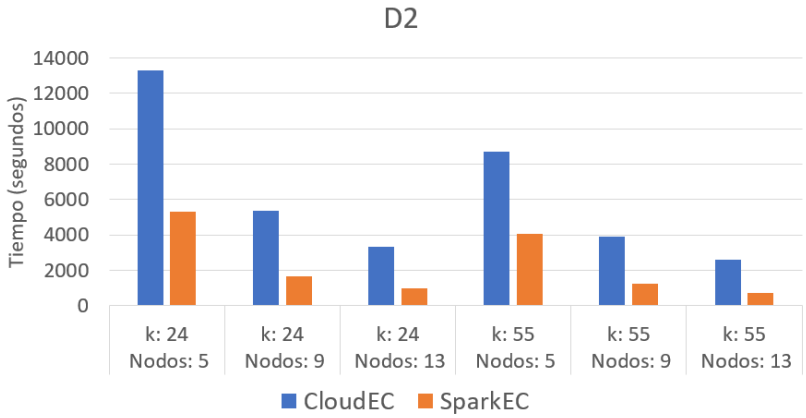

Figura 6.3: Resultados para D2

embargo, la mejora media sigue siendo significativa, rondando aproximadamente 2,6x (ver Figura 6.4). Esto significa que, incluso en el escenario más desfavorable, SparkEC es al menos el doble de rápido que CloudEC independientemente del valor de K.

- ***D4.*** Este es el escenario con menor longitud por secuencia y, sin embargo, tiene un número de secuencias de entrada elevado (muy similar a D1). El impacto de esto en el rendimiento es que, al tener una longitud de secuencia más reducida, no se generan tantos k-mers y, por tanto, el tiempo de ejecución se concentra más en tiempo de cálculo (i.e. de corrección) que en operaciones de shuffle a través de la red, las cuales impactan a ambos frameworks de forma muy similar. La conclusión de todo esto es que D4 es el escenario más favorable para SparkEC, alcanzado aceleraciones de hasta 11,8x respecto a CloudEC, tal y como se muestra en la Figura 6.5.

Cabe resaltar que en estas mediciones reportadas no se está incluyendo el tiempo ne-

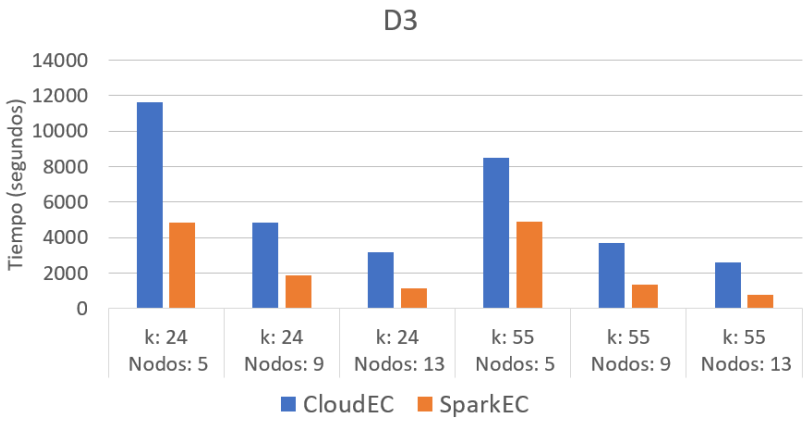

Figura 6.4: Resultados para D3

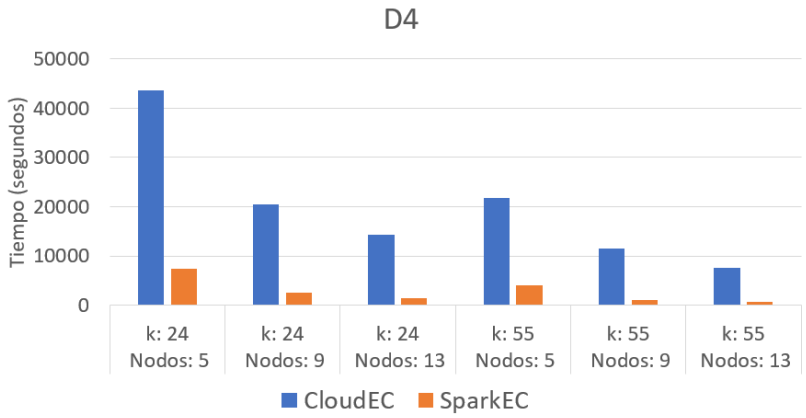

Figura 6.5: Resultados para D4

cesario para realizar el preprocesado de los datos de entrada en el caso de CloudEC. Esta información no se ha añadido al tratarse de tiempos poco significativos (no superiores a los dos minutos en el peor caso), aunque es importante tener en cuenta que este hecho penalizaría a CloudEC frente a SparkEC a medida que el conjunto de datos de entrada aumentase de tamaño.

A modo de resumen, la Tabla 6.3 presenta todos los resultados mostrados en las figuras anteriores de una forma agrupada para facilitar así su análisis.

6.3.1 Comparación fase a fase

El análisis comparativo de los tiempos fase a fase entre ambas herramientas se muestra a continuación únicamente para el conjunto de datos D2 y $K = 24$, ya que los resultados para el resto de conjuntos y valores de K presentaban un patrón muy similar. Cabe también

Dataset	Valor de K	#Nodos	Tiempo de CloudEC	Tiempo de SparkEC
D1	24	5	29.862	11.951
		9	13.697	4.429
		13	9.150	2.731
	55	5	21.909	9.693
		9	10.135	2.792
		13	6.216	1.785
D2	24	5	13.307	5.289
		9	5.351	1.659
		13	3.309	971
	55	5	8.688	4.035
		9	3.889	1.250
		13	2.594	700
D3	24	5	11.609	4.865
		9	4.831	1.885
		13	3.167	1.113
	55	5	8.502	4.892
		9	3.679	1.348
		13	2.616	756
D4	24	5	43.506	7.473
		9	20.383	2.484
		13	14.334	1.511
	55	5	21.723	3.987
		9	11.543	1.146
		13	7.617	648

Tabla 6.3: Tabla de tiempos (en segundos)

resaltar que se ha omitido la primera y última fase del algoritmo (i.e. PreProcess y PostProcess, ver Sección 5.1.2) ya que su cómputo es significativamente inferior al resto, centrando así el análisis comparativo en las partes más costosas del algoritmo de corrección.

Los resultados fase a fase para la corrección del conjunto D2 ($K = 24$) usando 5, 9 y 13 nodos se muestran en la Figura 6.6. Se puede observar que SparkEC consigue reducciones muy significativas en el tiempo de ejecución de ambas fases de filtrado, así como en la fase de corrección PinchCorrect. En SpreadCorrect, la fase de corrección más computacionalmente intensiva, la mejora de rendimiento es más limitada aunque significativa, en torno a un 40% de reducción en el peor de los casos (5 nodos) y aproximadamente un 60% en el resto. Esto se puede deber al gran volumen de datos que se generan en las operaciones de shuffle y que deben enviarse por la red, con lo que deben lidiar tanto Spark como Hadoop, relegando así el tiempo de ejecución del algoritmo de corrección a un segundo plano.

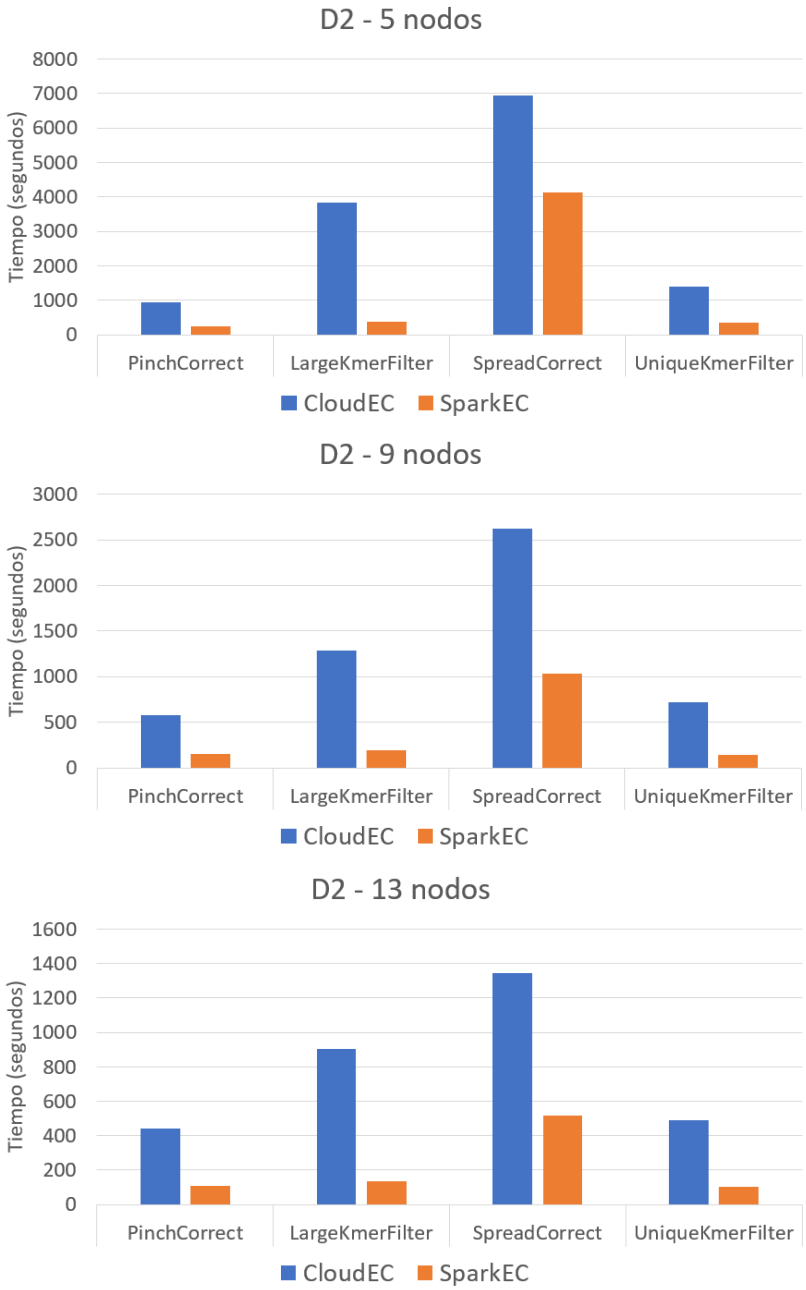

Figura 6.6: Resultados por fase para D2 ($K = 24$)

Capítulo 7

Conclusiones y trabajo futuro

Este TFG ha abordado el desarrollo de una herramienta paralela para la corrección de secuencias genéticas capaz de procesar un gran volumen de datos de forma eficiente y escalable. En este capítulo se analizan las principales conclusiones obtenidas durante la ejecución del proyecto, así como su relación con la titulación en la que se encuadra, y se mencionan posibles líneas de desarrollo futuras.

7.1 Conclusiones

El proyecto se inició tomando como punto de partida una herramienta existente para la corrección de errores en conjuntos de datos genómicos, CloudEC, implementada con el framework de procesamiento Big Data Apache Hadoop. A partir de su algoritmo de corrección subyacente, se ha desarrollado una nueva herramienta, SparkEC, implementada sobre el framework Apache Spark, que no solo es capaz de proporcionar resultados de corrección con la misma precisión que CloudEC, sino que lo hace consiguiendo mejoras significativas en el rendimiento en todos los escenarios evaluados. Esta evaluación se realizó en un entorno clúster usando hasta 13 nodos y procesando cuatro conjuntos de datos de entrada con diferentes características.

Siguiendo una metodología ágil, el diseño y desarrollo de la herramienta ha ido refinándose progresivamente a través de distintos Sprints para proporcionar el mejor rendimiento posible. Así, en las primeras versiones existían escenarios en los que la herramienta era incapaz de procesar algunos de los conjuntos de datos seleccionados, y a través de la aplicación de diversas optimizaciones se ha conseguido una solución eficiente al problema planteado, con un rendimiento cuatro veces superior a CloudEC de media, y hasta 11,8 veces superior en el mejor escenario. Por todo esto, se puede concluir que el software desarrollado cumple los objetivos para los que fue concebido. Además, en el proceso de reingeniería realizado en este TFG se hizo un rediseño completo aplicando buenas prácticas de Ingeniería del Software, de

cara a proporcionar una aplicación fácilmente modificable y extensible en el futuro.

Por otro lado, el desarrollo de este software me ha permitido no solo asentar todos los conocimientos adquiridos a lo largo del Grado y la mención, sino que también me ha permitido su ampliación. Esto ha sido posible gracias al uso de frameworks Big Data no estudiados en el Grado, y a poder trabajar en un entorno distribuido real (el clúster Pluton). Gracias a todo ello, este TFG ha contribuido favorablemente a mi formación.

Por último, resaltar que SparkEC es un producto software totalmente operativo que se encuentra disponible para su descarga en: `https://github.com/mscrocker/SparkEC`.

7.2 Relación con la titulación

Durante el desarrollo del proyecto se han aplicado diversos conocimientos, tanto del Grado en general como de la Mención en Ingeniería del Software en particular. Sin ir más lejos, he puesto en práctica los conocimientos adquiridos en la asignatura **Arquitectura del Software**, tanto para el análisis de la arquitectura de CloudEC y el diseño de SparkEC como para el estudio de las tecnologías utilizadas por ambas herramientas para su funcionamiento.

Por un lado, para el diseño se han aplicado a lo largo del desarrollo patrones y principios de diseño estudiados en la asignatura **Diseño Software**, con el fin de mejorar la calidad y mantenibilidad de la herramienta desarrollada. Además, los conocimientos adquiridos en la asignatura **Concurrencia y Paralelismo** han sido de gran utilidad a la hora de determinar las posibles optimizaciones a aplicar, y de comprender el estudio de sistemas paralelos. Respecto a la implementación de la herramienta, han sido de gran provecho los conocimientos acerca de programación estructurada adquiridos en las asignaturas **Programación I** y **Programación II**. Relacionado con lo anterior, también se aplicaron técnicas de validación estudiadas en **Validación y Verificación del Software**. Finalmente, respecto a las metodologías utilizadas, ha sido de gran ayuda conocer tanto el proceso de desarrollo de un sistema gracias a la materia **Proceso Software**, como las diferentes metodologías disponibles y su funcionamiento estudiadas en **Metodologías de Desarrollo**.

También es posible analizar la relación del presente trabajo con la titulación mostrando las competencias propias del título [25] que se han utilizado a lo largo de su desarrollo. De entre todas estas competencias destacan las siguientes:

- ***A4:*** "Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería."
- ***A12:*** "Conocimiento y aplicación de los procedimientos algorítmicos básicos de las tecnologías informáticas para diseñar soluciones a problemas, analizando la idoneidad y complejidad de los algoritmos propuestos."

- ***A13:*** "Conocimiento, diseño y utilización de forma eficiente de los tipos y estructuras de datos más adecuados a la resolución de un problema."
- ***A20:*** "Conocimiento y aplicación de los principios fundamentales y técnicas básicas de la programación paralela, concurrente, distribuida y de tiempo real."
- ***A28:*** "Capacidad de identificar y analizar problemas, y diseñar, desarrollar, implementar, verificar y documentar soluciones software sobre la base de un conocimiento adecuado de las teorías, modelos y técnicas actuales."
- ***A41:*** "Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos."

7.3 Trabajo futuro

Para comenzar, es importante destacar que SparkEC se ha desarrollado teniendo en cuenta el hardware disponible en la actualidad. Resulta relevante considerar así que algunas optimizaciones que no se plantearon en el desarrollo actual podrían cobrar sentido en un escenario futuro, en el que los equipos tengan a su disposición una mayor cantidad de memoria.

Una de estas posibles optimizaciones que podrían impactar favorablemente en el rendimiento se basa en almacenar simultáneamente en memoria todas las secuencias y k-mers que se utilizarán a lo largo de la ejecución, manteniéndolas sincronizadas entre sí. Esta optimización podría reducir considerablemente el número de operaciones shuffle que se realizan durante la ejecución, lo que probablemente implicaría una mejora significativa en el rendimiento global del algoritmo.

Por otro lado, también se podrían modificar las representaciones internas de las secuencias, tratando de utilizar algoritmos de compresión de datos u otras estrategias de implementación sobre ellas. Esto sería posible hacerlo sin demasiado esfuerzo por la forma en la que se ha diseñado SparkEC, permitiendo así implementaciones diferentes de las secuencias.

Finalmente, podría resultar interesante introducir nuevas fases de corrección o filtrado. Esto no representaría un problema, ya que la arquitectura *Pipe and Filter* utilizada por SparkEC responde perfectamente a la introducción de nuevas fases (o incluso a la alteración del orden de ejecución de las fases). A esto contribuye también el hecho de que las fases se comuniquen utilizando un formato común de datos.

Lista de acrónimos

BDEv *Big Data Evaluator.*

DAG *Directed Acyclic Graph.*

GFS *Google File System.*

HDFS *Hadoop Distributed File System.*

HSP *Hadoop Sequence Parser.*

IDE *Integrated Development Environment.*

LRU *Least Recently Used.*

MSA *Multiple Sequence Alignment.*

NCBI *National Center for Biotechnology Information.*

NGS *Next Generation Sequencing.*

RDD *Resilient Distributed Dataset.*

UML *Unified Modelling Language.*

YARN *Yet Another Resource Negotiator.*

Glosario

Bytecode Código intermedio, independente de la máquina y generado por los compiladores de determinados lenguajes de programación (Java, Erlang, Scala...) para ejecutarse posteriormente por el intérprete correspondiente.

Ejecución eager Paradigma de ejecución opuesto al paradigma lazy. En él, las operaciones se realizan tan pronto como se solicitan.

Ejecución lazy Paradigma de ejecución en el que el cálculo no se realiza en el primer momento que se solicita, sino que se pospone su ejecución, permitiendo dedicar el tiempo del procesador a otras tareas.

K-mer Subsecuencia de longitud K extraída de una secuencia de nucleótidos.

Nucleótido Molécula elemental que representa información en un ácido nucleico, ya sea de ADN o de ARN.

Secuenciación Proceso de lectura de secuencias genéticas para su posterior tratamiento o almacenamiento.

Timebox Técnica aplicada en metodologías ágiles de desarrollo de software, basada en establecer plazos fijos de tiempo en los que se deben hacer entregas, y aplazar la funcionalidad que no pueda implementarse en ese intervalo.

Bibliografía

[1] W. Chung, J. Ho, C. Lin, and D. T. Lee, "CloudEC: A MapReduce-based algorithm for correcting errors in next-generation sequencing Big Data," in *Proceedings of the 2017 IEEE International Conference on Big Data (IEEE Big Data 2017)*. Boston, MA, USA, 2017, pp. 2836–2842.

[2] The Apache Software Foundation, "Apache Hadoop." [En línea]. Disponible en: http://hadoop.apache.org

[3] S. Behjati and P. S. Tarpey, "What is next generation sequencing?" *Archives of Disease in Childhood-Education and Practice*, vol. 98, no. 6, pp. 236–238, 2013.

[4] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for Big Data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[5] X. Yang, S. P. Chockalingam, and S. Aluru, "A survey of error-correction methods for next-generation sequencing," *Briefings in Bioinformatics*, vol. 14, no. 1, pp. 56–66, 2012.

[6] S. Gnerre *et al.*, "High-quality draft assemblies of mammalian genomes from massively parallel sequence data," *Proceedings of the National Academy of Sciences*, vol. 108, no. 4, pp. 1513–1518, 2011.

[7] C. Chen, Y. Chang, W. Chung, D. Lee, and J. Ho, "CloudRS: An error correction algorithm of high-throughput sequencing data based on scalable framework," in *Proceedings of the 2013 IEEE International Conference on Big Data (IEEE Big Data 2013)*. Santa Clara, CA, USA, 2013, pp. 717–722.

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*. Incline Village, NV, USA, 2010, pp. 1–10.

[11] V. K. Vavilapalli *et al.*, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. Santa Clara, CA, USA, 2013, pp. 5:1–5:16.

[12] M. Zaharia *et al.*, "Resilient Distributed Datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. San Jose, CA, USA, 2012, pp. 15–28.

[13] "RDD programming guide." [En línea]. Disponible en: https://spark.apache.org/docs/latest/rdd-programming-guide.html

[14] "Hadoop Sequence Parser (HSP) library." [En línea]. Disponible en: https://github.com/UDC-GAC/hsp

[15] "BDEv: Big Data Evaluator." [En línea]. Disponible en: http://bdev.des.udc.es

[16] J. Veiga, J. Enes, R. R. Expósito, and J. Touriño, "BDEv 3.0: Energy efficiency and microarchitectural characterization of Big Data processing frameworks," *Future Generation Computer Systems*, vol. 86, pp. 565–581, 2018.

[17] "Guía de Scrum." [En línea]. Disponible en: https://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-Spanish.pdf

[18] M. O. Ahmad, J. Markkula, and M. Oivo, "Kanban in software development: A systematic literature review," in *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2013)*. Santander, Spain, 2013, pp. 9–16.

[19] "The C4 model." [En línea]. Disponible en: https://c4model.com

[20] "Especificación de los antipatrones de diseño de desarrollo." [En línea]. Disponible en: http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/204

[21] "EsotericSoftware/kryo: Java binary serialization and cloning: fast, efficient, automatic." [En línea]. Disponible en: https://github.com/EsotericSoftware/kryo

[22] E. W. Sayers *et al.*, "Database resources of the National Center for Biotechnology Information," *Nucleic Acids Research*, vol. 48, no. D1, pp. D9–D16, 2020.

[23] R. Leinonen, H. Sugawara, and M. Shumway, "The Sequence Read Archive," *Nucleic Acids Research*, vol. 39, no. suppl_1, pp. D19–D21, 2010.

[24] "Clúster Pluton." [En línea]. Disponible en: http://pluton.dec.udc.es

[25] "Facultad de Informática: Guía docente 2019/20." [En línea]. Disponible en: https://guiadocente.udc.es/guia_docent/index.php?centre=614&ensenyament=614G01&consulta=competencies&idioma=cast